CodeWarrior®
Learn Pascal Under Windows 95/NT

metrowerks

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Metrowerks CodeWarrior copyright ©1993–1996 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international

Metrowerks Corporation
2201 Donley Drive, Suite 310
Austin, TX 78758
U.S.A.

Canada

Metrowerks Inc.
1500 du College, Suite 300
Ville St-Laurent, QC
Canada H4L 5G6

Mail order

Voice: (800) 377–5416
Fax: (512) 873–4901

World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
America Online	keyword: `Metrowerks`
CompuServe	goto `Metrowerks`

Table of Contents

1 Introduction 15
- Pascal 15
- CodeWarrior Pascal 17
  - Programs and Projects 18
- The CodeWarrior Pascal IDE 18
  - Starting the IDE 19
- Creating and Running an Example Program 22
  - Creating the Source Program. 23
  - Creating the Project. 24
    - Command Shortcuts 27
  - Running the Program in Standalone Mode. 28
- On-Line Help 28
- Summary 28

2 Pascal: An Overview 29
- Starting the IDE 29
  - Help! 30
- Pascal Program Structure 31
  - Pascal Program Style 33
  - Pascal Program Execution 33
  - Statements 34
    - Simple Statements 34
    - Compound Statements 35
- Creating and Running the Program 35
  - Creating a Source File. 35
  - Integrating Sources and Resources 35
    - Stationery for Source and Library Correspondence 35
    - Combining Files to Create a Project. 36
  - Building an Executable Program 39
  - ... And Executing It 40
- Changing a Program 40
  - Opening the Source File. 40
  - Editing the Source 41
- Another Example Program. 42

  - Making the Program More Flexible 45
  - The Edit Window Revisited 46
- Example: Using Complex Data Types 47
  - Recycling Your Programs 53
- Example: Dealing with Files 60
  - The Bubblesort Algorithm 64
- Two Final Examples. 65
- Summary 68

3 CodeWarrior Pascal's Simple Data Types 69
- Pascal Programs 69
- Variables 74
- Naming Variables. 74
- Variable Declarations 77
- Constant Definitions 83
- Predefined Procedures for Input and Output 85
- Real Numbers 90
- Whole Number Types. 106
- Character-Based Types 122
- Logical Types 128
- Enumerated Types 141
- Summary 143

4 Procedures & Functions 145
- Procedures: General Features. 145
  - Procedure Definitions and Calls 146
- Pascal Procedures. 148
  - The Procedure Heading 149
  - The Declaration Section 149
  - The Procedure Body 150
  - Global and Local Variables 150
- Parameters. 154
    - Multiple Parameters 155
    - Formal Parameters and Arguments 156
  - Declaring and Using Parameters 157
  - Two-Way Parameters 158

Examples of Parameter Passing 162
Functions 165
  Defining Functions in CodeWarrior Pascal 166
  Calling a Function 167
  Parameters to Functions 170
Predefined Procedures and Functions 170
Predefined Routines in Standard Pascal 171
  FUNCTION ABS 171
  FUNCTION ARCTAN 172
  FUNCTION CHR 173
  FUNCTION COS 175
  FUNCTION EXP 177
  FUNCTION LN 178
  FUNCTION ODD 179
  FUNCTION ORD 181
  FUNCTION PRED 182
  PROCEDUREs READ and READLN 184
  FUNCTION ROUND 184
  FUNCTION SIN 185
  FUNCTION SQR 187
  FUNCTION SQRT 189
  FUNCTION SUCC 191
  FUNCTION TRUNC 192
  WRITE and WRITELN Procedures 194
Routines Predefined in CodeWarrior Pascal 194
  FUNCTION CONCAT 194
  FUNCTION COPY 196
  PROCEDURE DEC 198
  PROCEDURE DELETE 200
  PROCEDURE INC 201
  PROCEDURE INSERT 203
  FUNCTION LENGTH 205
  FUNCTION POS 206
  FUNCTION RAND 208
  FUNCTION SIZEOF 208
  PROCEDURE SRAND 211

FUNCTION UPCASE 211
Summary 212

5 CodeWarrior Pascal's Control Structures 215
Simple Repetition: The FOR Statement 215
Looping Backwards 220
Nested FOR Loops 222
Selection Constructs: The IF Statement. 226
Deciding Between Alternatives: The IF-ELSE Statement . . . 229
Working Through Alternatives: An Extended IF-ELSE Statement 231
Beware of the Missing ELSE 242
Selecting from Among Several Possibilities: The CASE **Statement. 246**
Iteration Constructs 252
The Impulsive Iteration Construct: The REPEAT Statement . 253
The Cautious Iteration Construct: The WHILE Statement . . 257
The Structure of Iteration Statements 262
A Comparison Example 267
Getting Around the Looping Rules 269
CYCLE 269
LEAVE 271
EXIT 273
HALT. 273
Summary 273

6 Procedures & Functions Revisited. 275
Calling Procedures and Functions. 275
Nested Routines 277
Program Design Issues 279
Reading Source Files During Compilation 283
Accessibility of Routines and Variables 286
Scope and Accessibility 296
Visibility 297
Recursion 300
The Flow of a Recursive Call. 303
Features of a Recursive Routine 303

    Another Example 305
    Iterative Solutions to Recursive Problems 311
  Summary 312

7 CodeWarrior Pascal's Structured Data Types 313

  Definitions and Declarations 314
  The ARRAY Type 315
    Defining and Declaring an Array 316
    Accessing Individual Array Cells 318
      Cell Index versus Cell Value 319
    Representing an Array 320
    Rules for Arrays 320
    Arrays as Parameters 321
    Operations on Arrays 322
    One More Example 323
    Strings 325
    Multidimensional Arrays 329
    Accessing Elements in Multidimensional Arrays 330
      Accessing Component Arrays 333
    String Arrays 334
      Three-Dimensional Arrays 337
    Arrays: A Final Point ... and Another Data Type 339
      The Subrange Type 339
  The RECORD Type 341
    Accessing Fields in a Record 342
    Structured Types As RECORD Fields 344
    Using WITH to Shorten Field Names 348
      Independent WITH Statements 352
      Nested WITH Statements 353
      Using Names for Variables and for Fields 353
    Arrays of Records 354
      Accessing Records in the Array 358
      The WITH Statement Revisited 358
    Operations on Records 359
    Variant Records 359
      Tagged and Untagged Variants 362

The OBJECT Type: A Preview 364
The SET Type 366
  Set Features and Restrictions. 369
  Representation of Sets. 370
  A Special Set. 370
  Set Operators 371
    Checking Whether an Element Is in a Set 372
    The Set Union Operator. 374
    Set Unions Involving the Null Set 377
    The Set Intersection Operator 378
    The Set Difference Operator 383
  Comparing Sets 390
    The Set Equality and Inequality Operators 390
    The Subset Operator 391
    The Superset Operator 391
Summary 393

8 Files 395

Text Files 396
Actions on Files 396
  Associating a Program and a Disk File 397
  Creating a New File 398
  An Example 402
  Checking the Outcome of Your File Operation 404
    A Safe File Creation Procedure. 406
  Opening an Existing File 408
    READ versus READLN when Dealing With Files 413
Example: Removing TABs From a File 417
    Reading Single Characters from a File 422
  Adding to An Existing File 422
  Operations on Text Files. 425
Typed Files 426
  Defining a Typed File 430
    Declaring a Typed File Variable 431
  Creating and Opening Typed Files 432
  Writing to Typed Files. 432

- Reading From Typed Files . . . 433
- Random Access to Typed Files . . . 433
- More Typed File Examples. . . . 441
  - Example: A File Containing Arrays. . . . 441
  - Example: Record and Set Types as File Components . . . 446
  - Example: Files Containing Enumerated Types . . . 450
- Untyped Files . . . 454
- Miscellaneous File Handling Routines . . . 459
- Summary . . . 459

9 Pointers . . . 461

- Static Memory Allocation . . . 461
  - Drawbacks of Static Memory Allocation . . . 461
- Dynamic Memory Allocation. . . . 462
    - The Program Heap . . . 462
- Pointers . . . 463
  - Defining Pointer Types . . . 465
  - Declaring Pointer Variables . . . 466
  - Allocating Dynamic Storage . . . 466
  - Accessing a Target Variable . . . 468
  - Assigning Values to Target Variables . . . 468
  - Don't Confuse Pointer and Target. . . . 470
  - Pointer Assignments . . . 471
  - Assigning Addresses to Pointers . . . 475
    - The Address, or Pointer, Operator . . . 475
- Pointer Pitfalls . . . 477
  - Reusing a Pointer . . . 477
  - Returning Storage to the Heap . . . 479
  - NIL: A Predefined Value for Pointers . . . 481
- Self-Referential Structures . . . 483
  - Linking Self-Referential Structures . . . 484
  - Building Linked Lists . . . 484
  - Adding at the Front of a Linked List . . . 485
    - InitLNode . . . 487
    - LLAtFront . . . 488
    - WriteList . . . 488

- Adding Elsewhere in a Linked List 488
  - Ordering a Linked List 492
- Example: Sorting Lines From A File 493
- Example: Using More Complex Nodes 503
- Summary 515

10 Units and Bits, Oh My 517

- Units : Public Interface and Private Implementation 517
  - The Interface Shows what a Unit Has to Offer 517
  - ... The Implementation Delivers on the Promise 520
- The CodeWarrior Pascal Units 521
  - Including the CodeWarrior Pascal Units 522
- Creating Your Own Units 524
  - Example: A Small Utilities Unit 524
- Bit Handling 528
  - Bitwise Negation: The NOT Operator 528
  - Bitwise Multiplication: The Bitwise AND Operator 532
    - Masks 534
    - Specifying a Mask 535
  - Bitwise Addition: The OR Operator. 538
  - Order of Precedence for the Bitwise Operators 542
  - Bitwise Exclusion: The XOR Operator 542
  - Toggling 542
- Shift Operators 543
  - The Left Shift Operator 543
  - The Right Shift Operator 547
    - Right Shifts and Powers of Two 550
    - IntBits (Value : INTEGER) : STRING 550
    - Precedence of the Shift Operators 551
- Summary 552

11 Object-Oriented Programming in CodeWarrior Pascal 553

- Traditional vs. Object-Oriented Programming 553
  - Example: Cataloging Written Materials 553
  - An Object-Oriented Approach 559
- Inheritance 560

Associating Actions With Object Types 561
Representing Objects 561
  Representing a Document 561
  Representing a Shape 562
Representing Inheritance 563
Methods. 568
Overriding Methods 574
  Assignments Between Different Object Types 579
  The Invisible Self Parameter 584
Summary 584

12 More on Object-Oriented Programming 585
Binding 585
Polymorphism 586
Flexibility 586
Beginning with a Collection of Numbers. 586
NrGrp Methods 587
Numerical Examination ... This is Not a Test 595
Possibilities 605
Taking an Object in a Different Direction. 605
Alea Iacta Est 612
A Final Exploration 622
Objects for Play and for Serious Play 632
Simulations 633
Summary 634

13 GUI Programming 635
Windows Programs 635
... And Windows Programming. 637
  Help for Windows Programs. 637
  The Program. 650
More Windows Elements 651

Index 665

1

Introduction

Welcome to CodeWarrior Pascal!

If this is your first experience with a programming language, you'll find that Pascal is an excellent language to learn, and the CodeWarrior Pascal programming environment should make your task a bit easier and more pleasant. If you've programmed before, but you're new to Pascal, you'll find it a clean and simple, but powerful language. If you're familiar with Pascal and are just curious, you'll find much to satisfy your curiosity in this implementation of the language.

Pascal

Pascal differs from most programming languages in that it was designed to *teach* programming, rather than to do programming. However, it has turned out to be very useful, and has been used to write all sorts of programs, even very large ones.

Pascal was created by Niklaus Wirth almost 30 years ago. Wirth had been on the committee that designed ALGOL, a very important language in the history of programming. As the committee worked, the ALGOL language became increasingly complex. Wirth became frustrated with this development, and decided to develop a language on his own.

Among other things, Wirth wanted to develop a language that was simple enough to teach as a first language. But he also wanted to develop a language powerful enough to make it possible to teach programming as a systematic discipline. Pascal fulfills both of these criteria very well.

As you'll see, Pascal is a small language that is easy to learn and easy to read. It is a *high-level* language, which means that its commands and data structures work at a level close to the one at which you would solve a problem. For example, you can create a data structure to represent a collection of information that belongs together. (A *data structure* is simply a template for storing a body of information in a particular way.)

In Pascal, this information can be all of the same type (for example, whole numbers) or of different types. For example, suppose you wanted to represent the following information about a recipe ingredient:

- ingredient's name
- amount of the ingredient
- food group
- number of calories
- cholesterol
- fat content
- protein content
- sodium content

You could create a data structure to represent this information. Pascal lets you specify such a data structure in a way that makes it easy to understand the representation.

The following Pascal code would describe a way of representing information about 20 ingredients. By reading the listing you should get a good idea of what is being represented. (Don't worry about the details of the syntax for now; you'll have plenty of opportunity to concern yourself with that later.)

```
TYPE FoodGroup = ( Unknown, EggsAndDairy, FruitsAndVegetables,
                   Grains, MeatAndFish);

     FoodUnit = ( Number, Ounce, Cup, Teaspoon, Tablespoon,
                  FluidOunce, Pinch, Gram, Dash);

     (* Describe a template for information
        about a single ingredient
     *)
     Ingredient = RECORD
                    Name : STRING;
                    Amount : REAL;
                    AmountUnit : FoodUnit;
                    Group : FoodGroup;
                    Calories : INTEGER;
                    Cholesterol, Fat, Protein : REAL;
                    Sodium : INTEGER;
```

```
         END;

(* Describe a template for info about 20 ingredients *)
IngredList = ARRAY [ 1 .. 20] OF Ingredient;
```

The first part of the listing creates a new type of information known as a **FoodGroup**. This type can take on only a small number of possible values — namely, the ones listed. Notice that you can use meaningful names for the values; you don't have to specify the values in numerical terms.

The next part of the listing creates another new type: **FoodUnit**. Variables of this type can contain values that represent commonly used units when discussing ingredient quantities.

The third part of the listing describes a data structure (named **Ingredient**) for storing information about a recipe ingredient. This structure contains several different items of information about the ingredient. For example, there is a slot in this ingredient record to store a name for the ingredient. There are also slots to store other types of information — including **FoodUnit** and **FoodGroup** values. You'll see how to create such data structures when you learn about Pascal's **RECORD** type.

Finally, there is a description of an **ARRAY**, a data structure for storing multiple items of the same kind. In this case, the items, or elements, are **Ingredient** records. As you can tell from the syntax, an **IngredList** array will store information about 20 ingredients.

The more you can represent a problem in terms familiar to you, the easier your task will be. You should always try to use the language to solve your problem, rather than modifying your problem to fit the language. Pascal makes it possible to make the language fit your problem.

CodeWarrior Pascal

CodeWarrior Pascal is, as you'll find, a very powerful programming environment, and a very solid implementation of Pascal. The environment includes an easy to use editor, a compiler, a debugger, and on-line help.

CodeWarrior Pascal provides several extensions to standard Pascal that make the language more powerful and also easier to use. The CodeWarrior implementation of Pascal has many of the best features of Borland's Turbo Pascal (arguably the most widely used implementation for DOS and Windows), and of Object Pascal (which is very popular in the Macintosh world).

For example, CodeWarrior Pascal makes it very easy to manipulate strings, which will come in handy when writing your programs. In the CodeWarrior Pascal environment, you can elect to use Borland's Turbo Pascal variant for doing input and output (I/O) — in particular, for working with files. Turbo Pascal I/O is very easy and widely supported.

CodeWarrior Pascal gives you access to Pascal's run time library (RTL) — a small, but well thought out — collection of predefined procedures and functions. In addition, CodeWarrior Pascal provides dozens of other procedures and functions, many of which are C functions that have been adapted for the CodeWarrior Pascal environment.

Programs and Projects

In CodeWarrior Pascal, you create programs by collecting all the components your program needs in a *project*.

A project will include library and interface files in addition to a source file that contains the raw form of your program. The library files contain precompiled versions of routines (procedures and functions) that you can use in your program. The interface files give you access to routines defined in CodeWarrior Pascal *units*, which are more specialized collections of data types and routines. Such units can extend the capabilities of your programming language and environment. For example, the units included with CodeWarrior Pascal give you access to dozens of C and C++ language routines. You can create your own units to carry out specific tasks.

And the best thing is that units can do this without cluttering up your program. By including a reference to a unit — for example, **USES Windows**, which you'll see in lots of the example programs — you effectively include the unit's contents in your source file, but in an invisible (and, hence, very neat and tidy) way.

You'll include these components by including them in your project, as you'll see later in the chapter.

The CodeWarrior Pascal IDE

The CodeWarrior IDE provides all the features and components you need to create Pascal programs for the Windows 95 and Windows NT environments. Actually, you can create Pascal programs for other platforms — for example, Macintosh 68K and PowerPC environments. You can also create programs in other languages — for example, in C/C++ and Java.

In this book, however, we'll be concerned only with Pascal for the 32-bit Windows 95 and NT environments. You can find out about all those other neat CodeWarrior features in the product documentation.

The IDE has it all: editor, compiler, linker, debugger, and browser — as well as the tools needed to manage your programming projects. You can create source files — text files that will be translated by the compiler and combined with other components by the linker.

The compiler will catch syntax —- programming grammar — errors, and will mark the location at which the error was detected. You can go back to the editor and make the required corrections. Because the compiler and editor can communicate in the integrated environment, you will be put at the error's presumed location in the source file. From the editor, you can also look at auxiliary materials (for example, header or interface files).

You can use the debugger to study your program — to find out where your program went wrong when it did what you said (rather than what you meant). With the debugger, you can trace through your program step-by step, check the values of particular variables during execution, follow procedure calls, and generally do the things you need to do to find out where the program made your mistake.

The browser will let you examine the elements in the programming language — for example, the relationships among the classes and elements defined for the language.

In this chapter, we'll take a very quick look at the IDE, and you'll build an example program. We'll examine the IDE in more detail in the next Chapter. So let's get going!

Starting the IDE

To start up the IDE, simply select it from the Metrowerks CodeWarrior group in your Startup List. Or, you can double click on the IDE icon in the folder containing your Metrowerks environment. When you start up the IDE, you'll see a window such as the one shown in Figure 1-1, although the IDE window may not be maximized upon startup. This window has three main components: the title bar, the CodeWarrior menu bar, and the edit window.

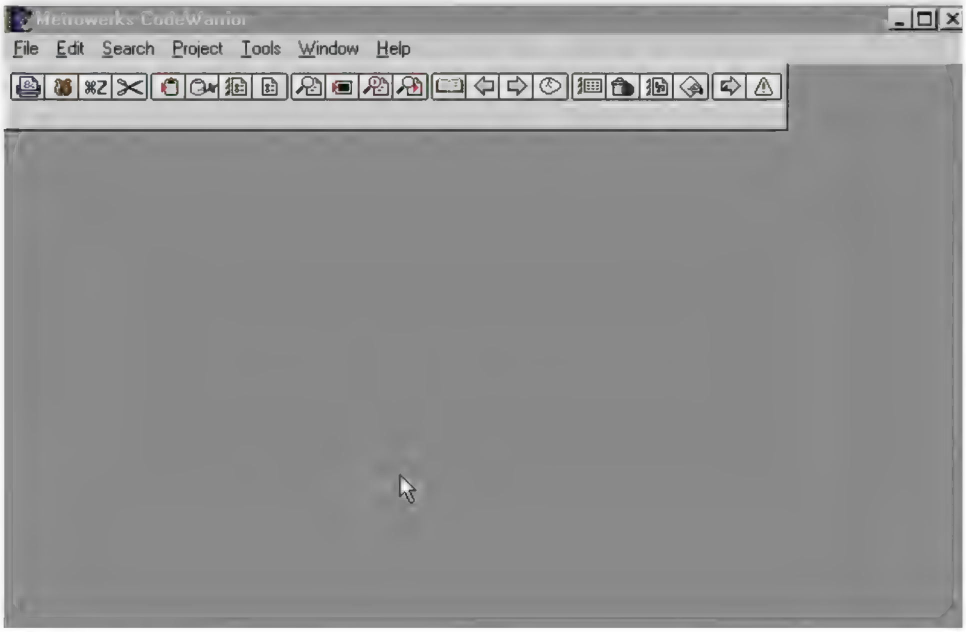


Figure 1.1 The main IDE window

You can set up your IDE and your programming (that is, language-specific) configuration through two selections from the edit menu: Preferences and Project Settings. The CodeWarrior *User's Guide* describes the options in the Preferences dialog box. The preferences allow you to configure your IDE projects to make it easier to see the different elements of your program.

For example, you can use different colors to distinguish comments from other parts of your source code. You can also have reserved words — those that are used as part of the language definition or implementation — displayed in a different color from the rest of the program.

In this book, we'll be concerned mainly with the Project Settings, although even these will be the same for most programs.

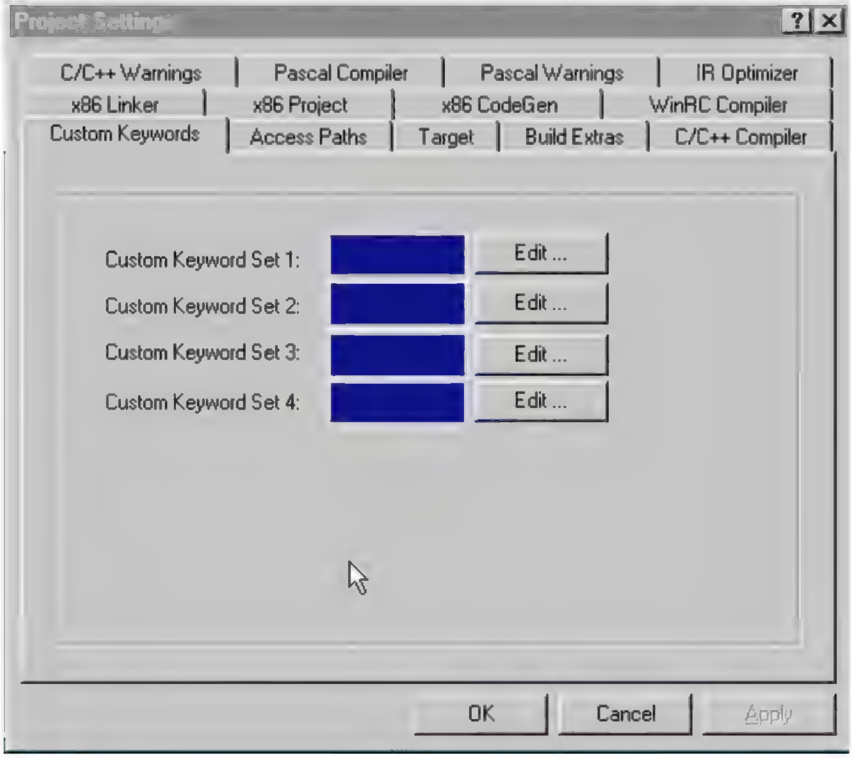

Project Settings
?
C/C++ Warnings
Pascal Compiler
Pascal Warnings
IR Optimizer
x86 Linker
x86 Project
x86 CodeGen
WinRC Compiler
Custom Keywords
Access Paths
Target
Build Extras
C/C++ Compiler
Custom Keyword Set 1:
Edit ...
Custom Keyword Set 2:
Edit ...
Custom Keyword Set 3:
Edit ...
Custom Keyword Set 4:
Edit ...
OK
Cancel
Apply

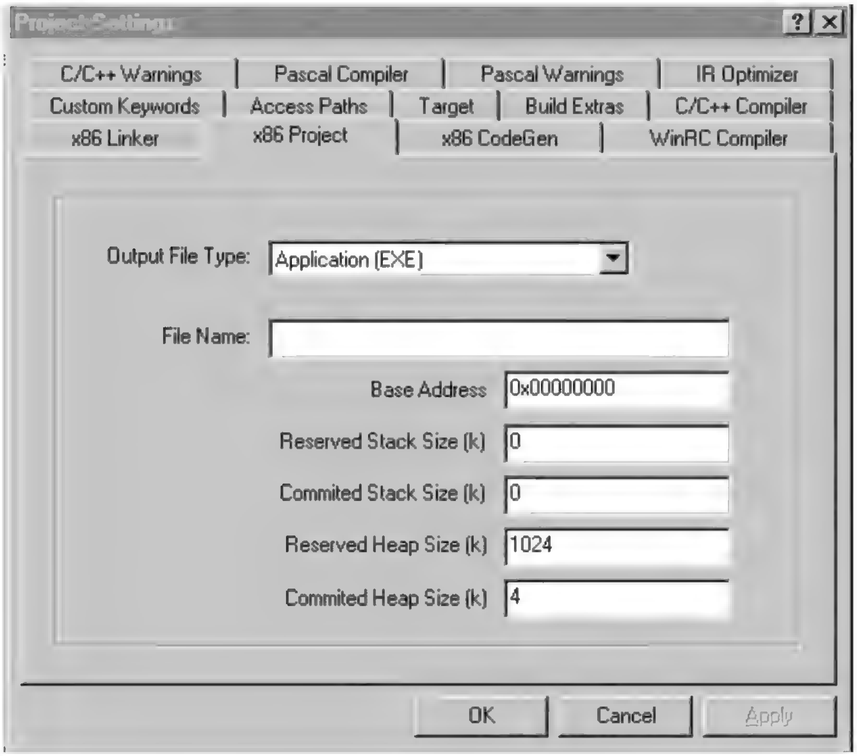
Figure 1.2 The Project Settings dialog box

Figure 1-2 shows the Project Settings dialog box you'll see when you select this menu item. To do this, click on the word Edit in the menu bar (or press the ALT and E keys at the same time). Then click on Project Settings or else use the arrow keys to highlight this item and then press ENTER to select it.

You can have Project Settings apply to all projects or just to specific projects. If you make changes to these settings in a "clean" IDE — that is, one without a project loaded — the settings will apply to all projects.

Any settings you change while you have a project open in the IDE will apply only to that project. After you close the project, the settings will revert to the values they had before you loaded the project.

The main dialog box for Project Settings includes several tabs, each of which gives you access to another set of options in a dialog box. In this chapter, we'll need to use just settings on just two of the tabs available in the Project Settings dialog box : x86 Project and x86 Linker.


Project Settings
C/C++ Warnings
Pascal Compiler
Pascal Warnings
IR Optimizer
Custom Keywords
Access Paths
Target
Build Extras
C/C++ Compiler
x86 Linker
x86 Project
x86 CodeGen
WinRC Compiler
OK
Cancel
Apply

Output File Type: Application (EXE)

File Name:

Base Address 0x00000000

Reserved Stack Size (k) 0

Commited Stack Size (k) 0

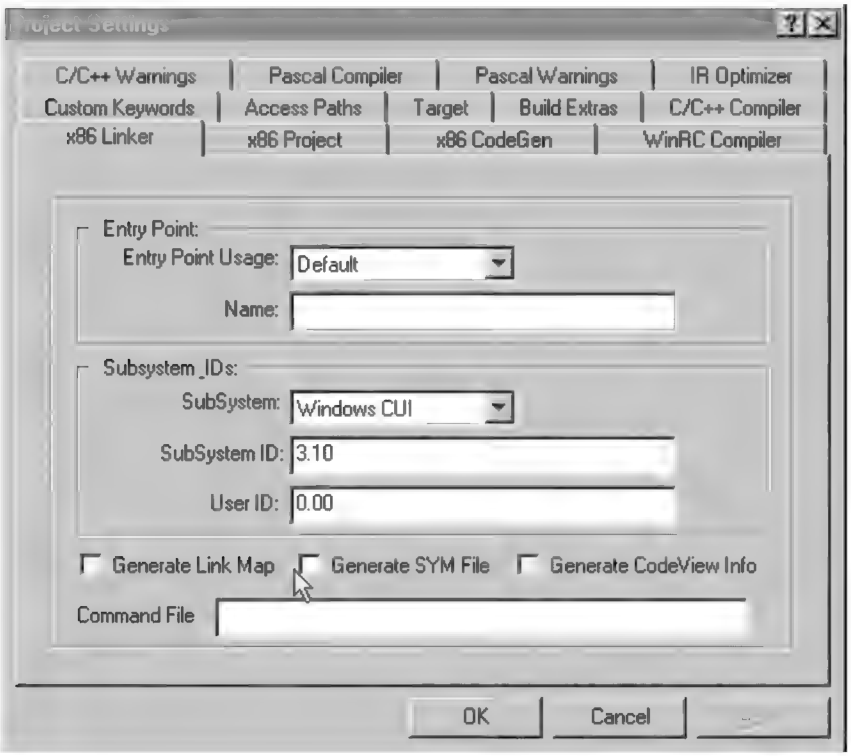
Reserved Heap Size (k) 1024

Commited Heap Size (k) 4

Figure 1.3 x86 Project dialog box

Figure 1-3 shows the x86 Project dialog box. In this box, you'll want to make sure the *Output File Type* is set to Application (EXE). If that's not already the setting, make this change before you load any projects. The second setting of interest in this box — *File Name* — is one you'll probably want to set for each project. *File Name* refers to the name you want to give your stand-alone program — that is, the file that you can execute even outside the IDE.

Unless you have special reasons for doing so, it's best to name your executable the same as your project. For example, the project you'll build later in this chapter is called 1-1.CWP. When I built it while writing this, I simply named the program 1-1.EXE.


C/C++ Warnings
Pascal Compiler
Pascal Warnings
IR Optimizer
Custom Keywords
Access Paths
Target
Build Extras
C/C++ Compiler
x86 Linker
x86 Project
x86 CodeGen
WinRC Compiler
Entry Point:
Subsystem IDs:
OK
Cancel

Entry Point Usage: Default
Name:
SubSystem: Windows CUI
SubSystem ID: 3.10
User ID: 0.00
☐ Generate Link Map ☐ Generate SYM File ☐ Generate CodeView Info
Command File

Figure 1.4 **The Linker dialog box**

Figure 1-4 shows the x86 Linker dialog box. For this chapter, the setting of interest in this dialog box is *SubSystem* under *SubSystem IDs*. In particular, for most of the programs in this book you should set this to Windows CUI — for Windows Console User Interface. This is a text-based window in which you can run programs that don't call on any of the Windows graphics capabilities. This makes the programs much simpler, and makes it easier to cover the fundamentals of the language without getting caught up in the details of the much more complex GUI (graphics user interface) that everyone has come to know and have

We'll look at some programs that do use the GUI later in the book.

Creating and Running an Example Program

Let's create, compile, and run an example program — just to see how it's done. Before you start, you should decide where you want to put your project and source files. You should create a separate folder for each project. On the CD, the example projects from this book are all in separate folders.

You also need to make sure that the program will be built for the Windows CUI (Console user interface), and *not* the GUI. You can specify this in the x86 Linker dialog box under Project Settings.

Creating the Source Program

The first step is to create the source program. To do this,

- Start the IDE
- Open the edit window by selecting New from the File menu. To drop down the File menu, either click on the word File on the menu bar or else press ALT and F at the same time. Once the menu has dropped down, either click on the New menu selection or else use the arrow keys until this sslection is highlighted, and then press ENTER.
- Enter the program shown in the following listing. (You can, of course, cheat — by getting the source file from the example project folder. But what will you do when you run out of example programs and you still don't know how to use the editor?)
- Save the source file under a name you'll remember easily — for example, one related to the project name. For example, you might save the source file as **1-1.PAS**, as I've named it in the project.

Listing 1.1

```
PROGRAM First;
BEGIN
  WRITELN ( 'Hello, fellow CodeWarriors.');
  WRITELN ( 'Press ENTER to end this program.');
  READLN;
END.
```

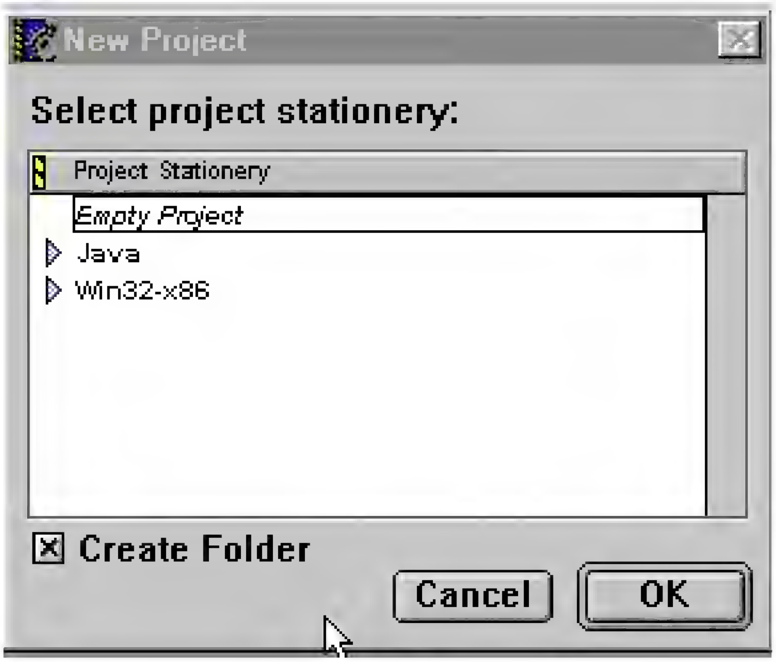
When you've finished entering the program, compare your text with the text in the listing. Are all the (single) quotes, parentheses, and semicolons present and in the right places? Are all the words — in particular, all the reserved words — spelled correctly? Does the program end with a period (and *not* a semicolon)?

Once you're confident you've entered the program correctly, save it as **1-1.PAS** (or as whatever you've decided to call it).

Creating the Project

You're now ready to start building the project. To do this, you may need to round up several types of files: library, interface, resource, and source files.

Fortunately, the IDE lets you cheat a bit, to make your task a bit quicker. To see how, select New Project from the File menu. You'll see a screen such as the one shown in Figure 1-5.



New Project

Select project stationery:

Project Stationery

Empty Project

Java

Win32-x86

☒ Create Folder

Figure 1.5 **The New Project dialog box**

Empty Project might seem like the choice that best fits your situation. However, you should click on the triangle next to **Win32x86**. This will give you a list such as the one shown in Figure 1-6. Double click on **Pascal Console Application** in this list.

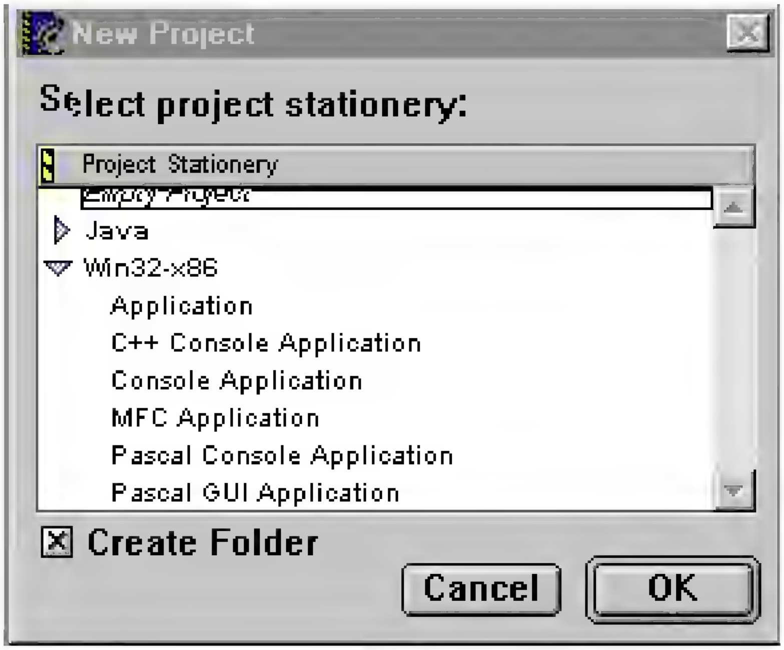
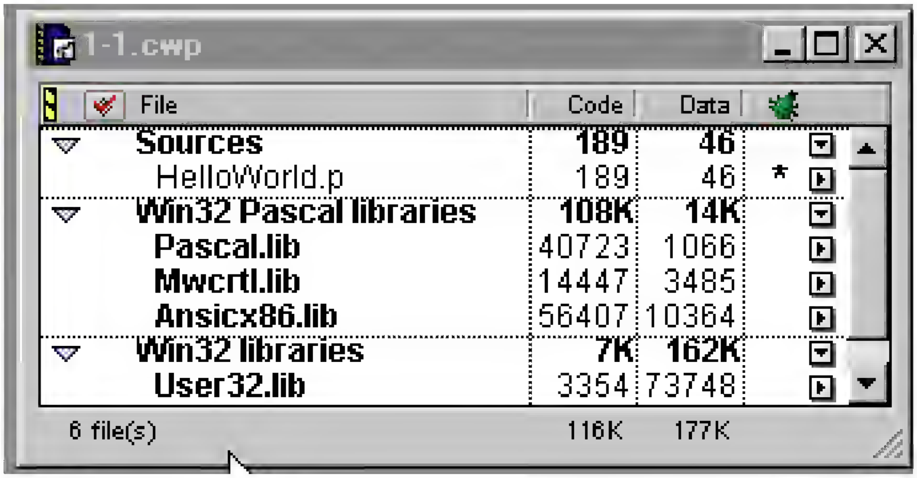

New Project
Select project stationery:
Project Stationery
Empty Project
Java
Win32-x86
Application
C++ Console Application
Console Application
MFC Application
Pascal Console Application
Pascal GUI Application
☒ Create Folder
Cancel
OK

Figure 1.6 Dialog box for selecting stationery for Win32x86 programs

You'll get a dialog box in which you can specify a name for your project. You can either click your way to the folder in which you want to place the project or you can type the path and project name in the **File Name** text box. If you specify **1-1.cwp** as your project name, you'll get a project window such as the one in Figure 1-7.



File	Code	Data	
Sources	**189**	**46**	
HelloWorld.p	189	46	*
Win32 Pascal libraries	**108K**	**14K**	
Pascal.lib	40723	1066	
Mwcrtl.lib	14447	3485	
Ansicx86.lib	56407	10364	
Win32 libraries	**7K**	**162K**	
User32.lib	3354	73748	
6 file(s)	116K	177K	

Figure 1.7 Window containing project stationery

This window contains the components needed for a sample project, including a source file (**HelloWorld.p**). You can use this project shell — known as the *stationery* — to create your own project folder.

To do this,

- Click on **HelloWorld.p**, to highlight it.
- Select **Add Files** from the **Project** menu.
- In the resulting dialog box, specify the path and name for your **1-1.PAS** source file.
- Click on **Add**.
- If you get an **Access Paths** message box, close it.

The project stationery should now include both **HelloWorld.p** and **1-1.PAS**. You've successfully added your source file to the project shell. The next step is to remove the **HelloWorld** program — which was serving as a placeholder. To do this,

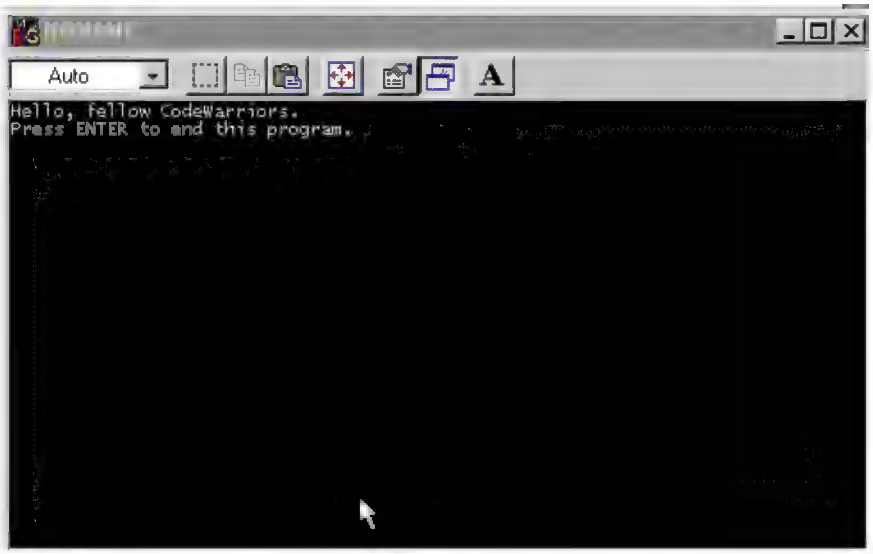
- Click again on **HelloWorld.p**, to highlight it.
- Select **Remove Files** from the **Project** menu.

The project shell should now contain only your source file, as well as the Pascal and Win32 libraries. You're ready to build your program. To do this, select **Make** from the **Project** menu. (You can also use the CTRL+M shortcut key to give this command.)

The compiler and linker will process your project. They'll report on their actions in the bar below the Toolbar (or below the menu bar if you have hidden the Toolbar).

If either of the program handlers finds any errors, they will report them in an error window. If you haven't made any typing or other errors, then these components will build an executable program named **1-1.exe** (or whatever name you specified). If you didn't specify a name in the Project Settings, then the default name — **NONAME.EXE** — will be used for the resulting program's name.

If your program compiles and links successfully, you can try it out right in the IDE. Select the **Run** command from the **Project** menu (or just use the CTRL+R shortcut keys). The program will execute in a small (but resizable) text-based window, as shown in Figure 1-8.



```
Hello, fellow CodeWarriors.
Press ENTER to end this program.
```

Figure 1.8 **Program executing in a console window**

Congratulations, you've just run your first CodeWarrior Pascal program!

Command Shortcuts

Many menu commands have shortcut keystrokes associated with them. These are keypresses that have the same effect as going through a series of menu selections -- except that they're much quicker. If you can remember them, the shortcuts can save you considerable time — a wetware example of the speed and memory tradeoffs that we read so much about in the trade journals and popular computing magazines.

Running the Program in Standalone Mode

You don't need to be in the CodeWarrior environment to run the program you've just created. To run your program as a standalone, exit from the CodeWarrior environment, and move to the folder containing your project's program. Double click on the icon for **1-1.EXE**.

On-Line Help

You'll find that there is very extensive on-line help available in the CodeWarrior environment. To access it, just click on Help in the IDE. From the menu you can either jump right into the online documentation or you can learn more about how to use CodeWarrior's help resources. You can even customize the Help environment.

Summary

In this chapter, you learned a bit about Pascal's heritage and goals, and also a few things about some of the language's features. In the next chapter, you'll get an informal survey of some of Pascal's elements. You'll also'll learn more about the IDE.

2

Pascal: An Overview

This chapter includes several example programs that illustrate some of Pascal's major constructs. These examples will give you a feel for how Pascal programs look and for the kinds of features Pascal offers. The examples also will provide a general background for the more detailed presentation of the language in subsequent chapters.

Don't worry about the details of the programs for now. The intent is to give you some exposure to Pascal. By running and modifying these programs on your computer, you'll also get an introduction to the CodeWarrior IDE. You'll learn about some of the more common commands, and about the capabilities available in the IDE. In the second part of the chapter, you'll find more information about the IDE. You may want to skim that now, and refer to it as you need to. You'll also find lots of information about the IDE in the documentation included with your CodeWarrior package.

To get the most out of this chapter, you'll need to run the programs and try making the editing and other changes suggested. You'll have to work with programs without knowing anything about Pascal. Fortunately, Pascal's syntax makes it possible to write the program in such a way that the source code can make it clear what the program does. This means you should be able to guess what the program does just by reading it properly.

Starting the IDE

To start the CodeWarrior IDE, simply select it from the Metrowerks CodeWarrior group or, if you did a customized installation, from the program group to which you added the CodeWarrior components.

The main IDE screen has five sections (from top to bottom), as shown in Figure 2-1:

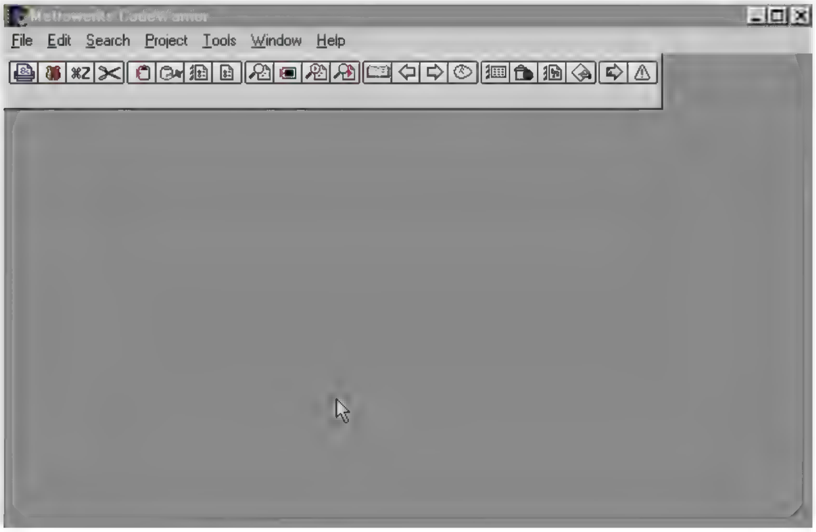


Figure 2.1 The main IDE window

- *Title Bar* : This contains the application's name, a *control menu* in the upper left hand corner, and buttons for maximizing, minimizing, and closing the application. You can move the IDE window by dragging the title bar. The control menu lets you move or resize the menu, as well as maximizing and minimizing it. The title bar behaves just as other Win95 title bars do.

- *Menu Bar* : This lists the menus available in the IDE. Several of the menus let you make further selections within the menu's general structure. The elements of this line remain the same throughout your session. You can select, or *pull down*, one of these menus, to get access to a list of selections.

- *Toolbar* : This contains icons to give you one-step access to some of the more commonly used commands. See the *User's Guide* for more information.

- *Status Bar* : This is where compiler and linker messages are displayed while a project is being processed to create a program.

- *Client Window* : This is where you will work if you use the IDE editor to create or edit a source file.

Help!

With any application, it's always a good idea to become familiar with its online help facilities. The IDE Help menu gives you access to the online documentation through three different paths:

- The *Contents* command puts you right at the start of the documentation for the IDE. You can move around in this file just as you would in any other document.
- The *Keys* command lets you narrow your search to a class of documents and also lets you search for specified text in some or all of the online documentation. Before you can search for information about specific words, however, you need to let the help facility create a database of all words referenced in the documentation. The Find Setup Wizard does this. To get access to this wizard, click on the Find tab in the dialog box you get when you select help by keys.
- The *How to Use Help* command lets you learn how to find information in the online documentation, and also how to copy this information to a file.

The online help is not language specific.

Pascal Program Structure

We'll use the following example to begin our study of Pascal programs. At this point, our main concern will be with the overall structure and features of Pascal programs.

Listing 2.1

```
(* Write a greeting and a value to the screen. *)

PROGRAM SecondExample;

VAR   Name : STRING;
      Ch : CHAR;

  PROCEDURE SayHello;
  BEGIN
    WRITE ( 'Hello there, and welcome to the CodeWarrior IDE, ');
  END; (* SayHello *)

BEGIN  (* Main program body *)
  Name := 'whoever you are'; (* Assign a value to Name *)
  SayHello;
  WRITELN ( NAME);
  WRITELN ( 'Press ENTER to end program.');
  READLN ( Ch);
END.
```

You can find the source code for program **SecondExample** in file **2-1.PAS**, which is part of project 2-1.CWP However, I urge you to type in the program yourself. It's short, but you'll still get some opportunity to work with the IDE editor.

This program initializes a **STRING** variable with the value "whoever you are," and then writes a greeting to the screen. The parting message tells you to press ENTER to officially end the program. Unless there is a special reason for not doing so, all programs in this book will wait until you press ENTER. This enables the programs to stay in the display window until you've finished studying them.

The structure of program **SecondExample** illustrates the three main components of a Pascal program:

- A *heading*, which will specify the program's name, and will possibly include additional information. In the example, this is the line that begins with the reserved word **PROGRAM**. (Reserved words are part of the Pascal language definition, and can't be used for other purposes in a Pascal program.) The program heading is required.
- Optional *declarations*, which may specify variables or constants. You can also declare "actions," by defining procedures or functions to carry out a specified task. Once included in a declaration section, these routines can be called at appropriate points in the program. Much of the challenge of learning Pascal is to learn how to write good procedures and functions. In the example, one variable (**Name**) is declared and one procedure (**SayHello**) is defined. Declarations are optional, although it's tough for a program to do anything interesting without using at least some variables or routines.
- A *body*, which contains the main program instructions, or statements. In the example, the program body contains five statements, including a call to **SayHello** and calls to procedures **WRITE**, **WRITELN**, and **READLN**, which are predefined in Pascal. The program body is also required.

The declaration section of a program is very important, as you'll learn. Part of the secret of successful programming is to determine the most appropriate way to represent your problem. This means specifying data types to store the problem's components and values; it also means specifying procedures and functions to carry out the actions needed to produce the solution.

You'll find that a problem that uses the most appropriate data types will be much easier to solve than one that uses less appropriate ones. In fact, if you do the declarations properly, you'll sometimes find that the main program is a very simple sequence of procedure and function calls.

A Pascal program body — that is, the *main program* — starts with the reserved word **BE-**

GIN, and finishes with the reserved word **END**. This last word *must* be followed by a period in Pascal. Physically, the program body is the last component in the source file.

Any declarations — that is, variable, procedure, and function definitions — in the program must precede the **BEGIN** that starts the program body.

TIP: A program must have a period (.) after the END that terminates the program.

Pascal Program Style

You can include comments — notes to yourself or to other readers — in your Pascal source code by putting them between (* and *) or between { and }. (It's a good idea to be generous with comments in your programs.)

Pascal is quite flexible in letting you use space and indentation in your programs. For example, Pascal compilers don't care whether you leave spaces between a procedure name (such as **WRITE**) and the left parenthesis that follows it. Nor does it matter whether you leave a space between the left parenthesis and the material following it. (You can't put a space in the middle of a word, however.)

You'll find such spaces in the examples, because it makes the source code a bit more open and easier to read. You can also use indentation to make it easier to differentiate various portions of a program. For example, in the file **2-1.PAS**, statements in the program body and in procedure **SayHello** are indented 2 spaces more than the surrounding material.

Whether you imitate the comment, spacing and indentation conventions used here doesn't matter; you should, however, develop your own style and use it consistently.

Pascal Program Execution

In a Pascal program, execution begins at the start of the program body, and continues (generally with digressions) to the last statement in the body. Because of the way Pascal programs execute, the program body is often a good starting point when reading the source code. Remember, the program body — or the main program — comes at the *end* of the program source file.

The main program in **SecondExample** initializes **Name** to a specific value. The **:=** is Pascal's *assignment operator*. This operator stores the value to its right (“whoever you are" in the example) in the variable (storage slot) specified to the operator's left (**Name**).

After this, the program calls procedure **SayHello** and also calls the **WRITELN** and **READLN**, procedures, which are predefined in Pascal. These calls are *statements* — particular instructions to the system.

A *procedure*, or *function*, call is an instruction to begin executing the routine named in the call. Thus, a call to **SayHello** transfers control (temporarily) to that procedure. The individual statements in the procedure are executed, after which control reverts back to the main program (in this case).

Execution then continues from the point after the call. Thus, in **SecondExample**, after **SayHello** has finished, the value of **Name** is written by the call to **WRITELN**.

Statements

Within the **BEGIN** and **END** boundaries, a program or procedure body consists of statements. In Pascal, you can have *simple* or *compound* statements.

Simple Statements

A simple statement generally represents a single instruction, and accomplishes a single task. For example, an assignment and a procedure call are both accomplished in simple statements.

In the program, procedure **SayHello** consists of a single simple statement — a call to the predefined procedure, **WRITE**. The calls to **SayHello**, **WRITELN**, and to **READLN** in the main program are also simple statements.

Each of the following is a simple statement:

```
WRITE ( 'Hello');

MyValue := MyValue + 35;

SayHello;
```

The first statement says to write the word “Hello" (without the quotes) to the screen. The statement is a procedure call, which is effected by simply naming the procedure to be called. **WRITE** is a predefined procedure that is available for use in your program. This procedure is an instruction to write something to the standard output device (usually the screen).

The material to be written is specified as one or more arguments, or parameters, to **WRITE**. An *argument* is an item of information specified for use in a procedure or function. The information is passed into the routine (and may also be passed back out), and is used within the body of the routine.

In the call to **WRITE**, the material within the parentheses is a string argument. The single quotes around the argument identify the argument as a *string literal*, or *constant*. You'll find more information about **WRITE** and related routines in Chapter 3.

The second statement adds 35 to the current value of a variable named **MyValue**. Recall that the **:=** is Pascal's assignment operator. The value to the right of the operator is stored in the variable specified to the left of the operator. Thus, the new value of **MyValue** becomes 35 more than its old value.

The third line is simply a call to the procedure **SayHello**. When this statement is encountered, the current routine (which may be the main program or another routine) stops executing temporarily, while **SayHello** executes. You'll learn more about procedure calls and program execution in Chapters 4 and 6.

Compound Statements

A *compound statement* actually consists of multiple statements, which may themselves be either simple or compound. Compound statements in Pascal are bounded by a **BEGIN** and an **END**. You'll find compound statements particularly useful when you start using Pascal's control constructs — for deciding what your program should do next. The first example program doesn't contain any compound statements, but you'll see them soon.

Creating and Running the Program

In Chapter 1, you saw that the main steps in creating an executable program were to create the source code, integrate this with any necessary libraries and other resources, and translate any components that need it into a form more palatable to the operating system.

Creating a Source File

To create the program's source file, select **New** from the **File** menu. This will put you in the edit window. Enter the text for program **2-1.PAS**, and save it under this name.

The *CodeWarrior User's Guide* gives you all the details about using the editor (and the IDE, for that matter). If you're even a little familiar with Windows-based text editors, you should have no trouble using the one in the IDE.

Integrating Sources and Resources

Even though your program may be a little gem of perfection, it will need a little help from some friends before it can be turned into something that will actually do the work you intended for it.

In practice, this means it needs to use routines and data structures made available from other files, such as libraries and interfaces. These helpers need to be made available for when your file get processed.

Stationery for Source and Library Correspondence

You can collect the helper files manually, using the **Add Files** command on the **Project** menu. This is a straightforward process, but it can get tedious and time consuming after a while.

Fortunately, the CodeWarrior Pascal environment includes stationery files to simplify your task. A *stationery* provides a template for a project. This template includes all the library and other files a program needs in order to compile and link. The stationery even includes a source file, which makes it a complete project. The source file serves as a placeholder for your source file.

You can use the stationery to create a new project. All you need to do is replace the source file with your program file. After you make the replacement, most programs will be ready to compile and link. A few of the example programs, however, will require you to add some other files. We'll look at a couple of examples in this chapter.

Combining Files to Create a Project

To create a new project, do the following:

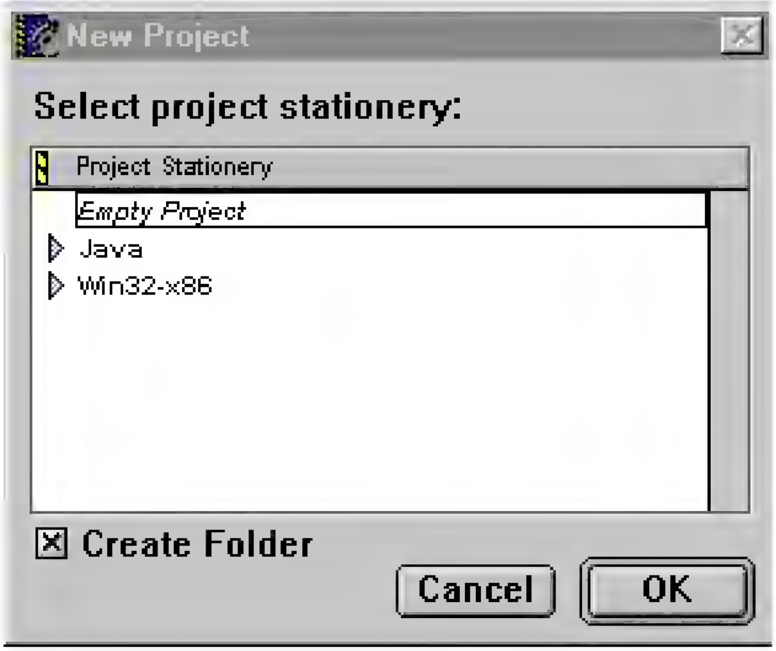
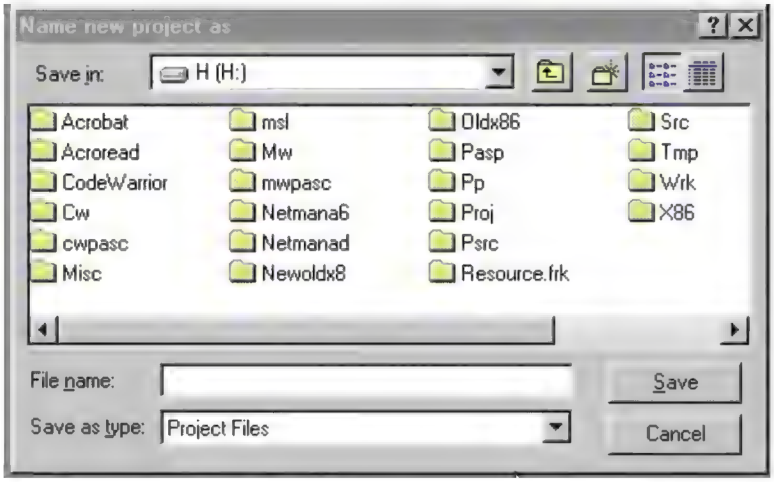


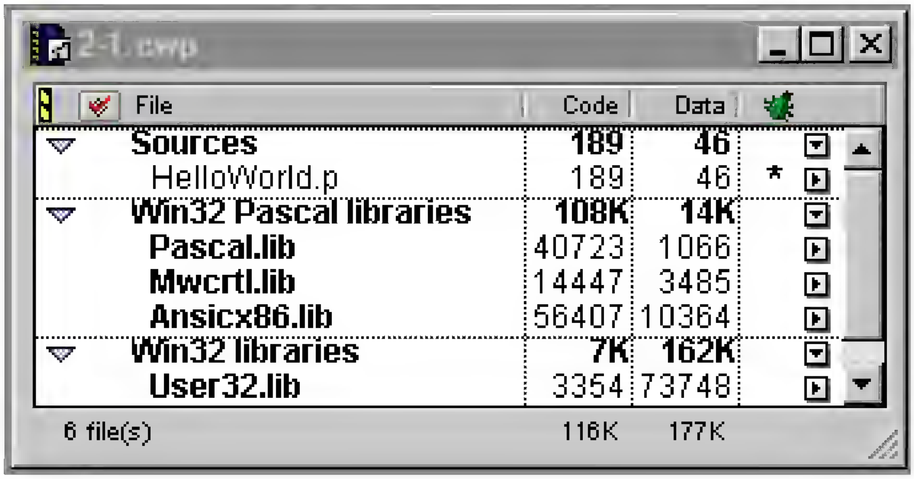
Figure 2.2 The New Project dialog box


Name new project as
Acrobat
Acroread
CodeWarrior
Cw
cwpasc
Misc
msl
Mw
mwpasc
Netmana6
Netmanad
Newoldx8
Oldx86
Pasp
Pp
Proj
Psrc
Resource.frk
Src
Tmp
Wrk
X86
File name:
Save
Cancel

Save in: H (H:)

Save as type: Project Files

Figure 2.3 The Name New Project dialog box

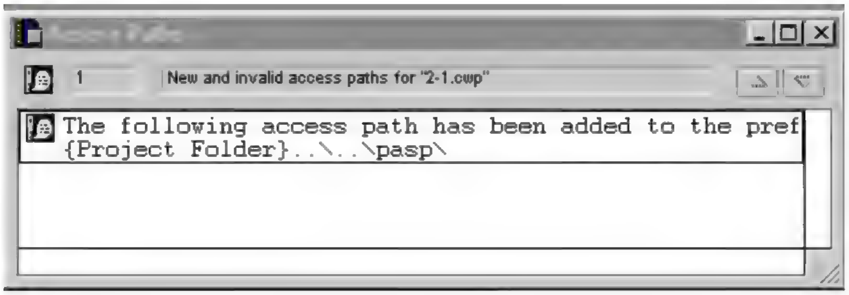


File	Code	Data
Sources	**189**	**46**
HelloWorld.p	189	46
Win32 Pascal libraries	**108K**	**14K**
Pascal.lib	40723	1066
Mwcrtl.lib	14447	3485
Ansicx86.lib	56407	10364
Win32 libraries	**7K**	**162K**
User32.lib	3354	73748
6 file(s)	116K	177K

Figure 2.4 Project stationery screen

- Select the **New Project** command from the **File Menu**.
- In the **New Project** dialog box (see Figure 2-2), select **Win32x86** as your project stationery. To do this, single-click on the triangle next to **Win32x86**.
- On the list that drops down, double click on **Pascal Console Application**.

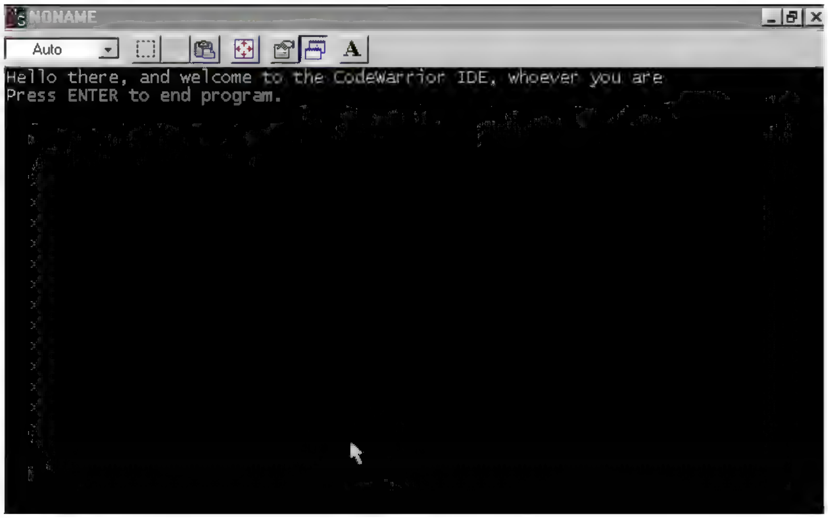
- In the **Name new project as** dialog box (see Figure 2-3), either type a project name in the **File Name** text box or double click your way to the folder you want to use for the project, and then specify a project name. Your project folder will be placed in the specified folder. CodeWarrior Pascal project names should have the extensions **CWP**, as in **2-1.CWP**.
- Click on **Save** once you have entered the **File Name** value. Figure 2-4 shows the screen after the project name has been specified. This is the stationery for your project.
- Move the mouse cursor to the dotted line under **HelloWorld.P**, and click once on the line. The **HelloWorld.P** file name should be highlighted. If the **Win32 Pascal libraries** group heading is highlighted instead, you clicked too low. Move the mouse cursor slightly toward the top of the window, and single click again — until you get **HelloWorld.P** highlighted.
- Now, select the **Add Files** command from the **Project** menu.
- In the **Select files to add** dialog box, either enter the name of your source file, including any necessary path information.
- After typing the name, press ENTER or click on **Add**. Depending on where your source file is and on how the access paths for your project are set up, you may get a message such as the one shown in Figure 2-5.
- If you get an **Access Paths** message box, you can either close it or ignore it. If you choose to ignore it, click on the stationery window, which should be in the background behind the **Access Paths** box. This will make the stationery window the active one. Once you've dealt with (or not dealt with) any such message box, you'll see the stationery for the project again. This time, however, your source file will be highlighted.
- Next, single-click on **HelloWorld.P**, to get it highlighted again. Be careful, however. Make sure it's the only file highlighted. Sometimes clicking on this file causes other files to be highlighted as well. If that happens, you may need to double click on **HelloWorld.P**, to open this file in the editor. If you do that, click on the close box (the X in the upper right corner of the window) immediately. You should now be able to highlight just **HelloWorld.P**.
-
- Once just this file is highlighted, select the **Remove Files** command from the **Project** menu. The file should disappear from the stationery list, so that only **2-1.PAS** is listed under the **Sources** group.
- Sit back and relax, since you've created the new project, and it's all ready to be built into a running program.



New and invalid access paths for "2-1.cwp"

```
The following access path has been added to the pref
{Project Folder}..\..\pasp\
```

Figure 2.5 Access path message



```
Hello there, and welcome to the CodeWarrior IDE, whoever you are
Press ENTER to end program.
```

Figure 2.6 Program running in a console window

Building an Executable Program ...

Once you've got the project file's stationery in the form you want, a running program is just two mouse clicks away. Select the **Make** command from the **Project** menu (or use the Ctrl+M shortcut).

The compiler and linker will report their progress in the status bar just under the toolbar. Generally, the messages will zip by, and you'll probably get little more than a headache trying to watch them.

If there are neither compiler nor linker errors, the IDE will look just as it did before you gave the **Make** command.

... And Executing It

If things went as planned, you now have a brand new **EXE** file sitting on your computer. You don't need to leave the IDE to see how your program works, however. Just select the **Run** command from the **Project** menu (or use the Ctrl+R shortcut).

Your program will execute in its own console window, as shown in Figure 2-6. You can move, maximize, and minimize this window, just as you can do with other windows.

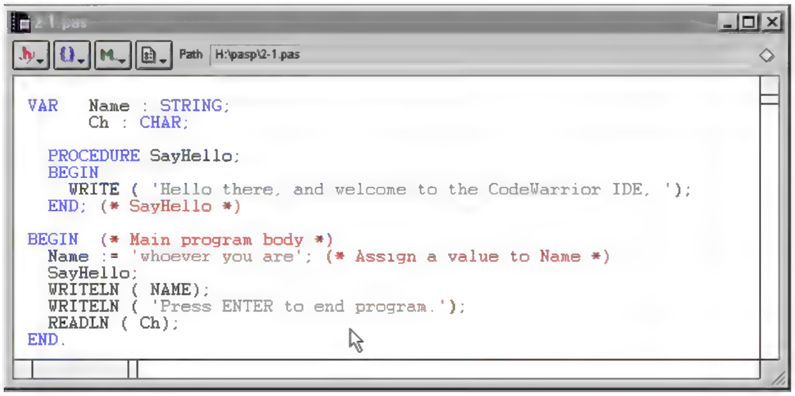
Changing a Program

If you read the program very carefully, you may have noticed that both **Name** and **NAME** are used in the program. This discrepancy does not seem to cause any problems.

This is because Pascal is *not* case sensitive. In Pascal, **Name** and **NAME** are considered equivalent. In this book, we'll use all uppercase characters when writing words that are predefined in Turbo Pascal.

Opening the Source File

For consistency, and also to get a little practice with the editor, let's change **NAME** to **Name** in the source file. You can access the file in either of two ways: with or without a project open.



```
VAR   Name : STRING;
      Ch : CHAR;

  PROCEDURE SayHello;
  BEGIN
    WRITE ( 'Hello there, and welcome to the CodeWarrior IDE, ');
  END; (* SayHello *)

BEGIN  (* Main program body *)
  Name := 'whoever you are'; (* Assign a value to Name *)
  SayHello;
  WRITELN ( NAME);
  WRITELN ( 'Press ENTER to end program.');
  READLN ( Ch);
END.
```

Figure 2.7 **Editing a source file from the IDE**

If your project is open — that is, if the project stationery window is accessible — you can simply double click on **2-1.PAS** in the list. The file will open (as in Figure 2-7), and you can edit it.

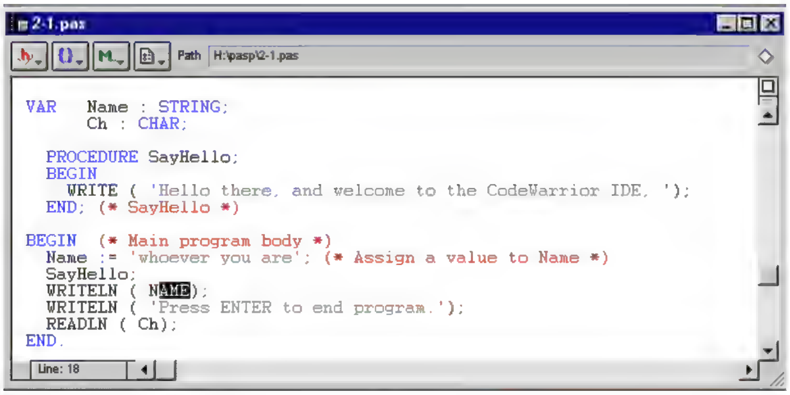
If you don't have the project open, you can get the file by selecting the **Open File** command in the **File** menu, and then selecting or specifying the file you want to edit. Note that you're selecting the **Open File** command, and not the **Open** command, which opens a project. (Actually, you could do it that way too. If you open the project, you find yourself in the previous case.) If you open the file directly — that is, not through a project stationery — you'll need to specify the entire path name for the file.

Editing the Source

Since the file **2-1.PAS** is small, you can easily move to the specified word by using the arrow keys on the numeric keypad. (Make sure NumLock is off, however.) You can also select the entire word or just the last three letters. See the *CodeWarrior User's Guide* if you need instructions on how to do this.

To make the desired change, do any of the following:

- Move the cursor to the space between the **N** and the **A** in **NAME**. Press the Del key three times — to delete **AME**. Then type **ame**.
- Move the cursor to the spot immediately after the **E** in **NAME**. Press the backspace key three times, then write **ame**.
- Move the cursor to the space between the **N** and the **A** in **NAME**. While holding down the left mouse button, slide the cursor to the end of the word — that is, over **AME**. Once you're done, release the mouse button. The last three letters of **NAME** should be highlighted, as in Figure 2-8. Just start typing the corrected version. The current version will disappear as soon as you begin.



```
VAR   Name : STRING;
      Ch : CHAR;

  PROCEDURE SayHello;
  BEGIN
    WRITE ( 'Hello there, and welcome to the CodeWarrior IDE. ');
  END; (* SayHello *)

BEGIN  (* Main program body *)
  Name := 'whoever you are'; (* Assign a value to Name *)
  SayHello;
  WRITELN ( NAME);
  WRITELN ( 'Press ENTER to end program.');
  READLN ( Ch);
END.
```

Figure 2.8 Making editing changes in a source file

Congratulations, you've made your first editing changes to a CodeWarrior program!

To end your example session, you just need to click on the **Close** box on the source file's title bar. Because you've made changes to the program file, you'll be asked whether to save the changes. If you want to keep the program with your changes in it, click on **Save**; if you want to keep the original version of the program, select **Don't Save**. Either of these commands will get you out of the file and out of the editor. To stay in the editing environment, click on **Cancel** in the dialog box.

Another Example Program

Let's look at another example program, so you can see some more of Pascal's features, and so you can learn a bit more about the CodeWarrior environment. The following listing shows the program

Listing 2.2

```
(* Program to generate random values.
   Program illustrates some Pascal control constructs.
*)

PROGRAM LoopExample;
USES Windows;
```

```
VAR   Count, RandSeed : INTEGER;
      Ch : CHAR;

   (* Generate random integers, and check whether
      certain of them are even or odd.
  *)
  PROCEDURE HandleRandom ( Count : INTEGER);
  VAR Value : INTEGER;
  BEGIN
    (* Get a random integer between 0 and 25000 *)
    Value := RAND MOD 25001;
    (* For every third value, tell whether it's even or odd *)
    IF Count MOD 3 = 0 THEN
    BEGIN
      WRITE ( Value : 5, ' is ');
      IF ODD ( Value) THEN
        WRITELN ( 'odd')
      ELSE
        WRITELN ( 'even');
    END
    ELSE (* for the other cases, just write the value *)
      WRITELN ( Value : 5);
  END; (* HandleRandom *)

BEGIN  (* Main program body *)
(*
  WRITE ( 'Integer seed for the random number generator? ');
  READLN ( RandSeed);
  SRAND ( RandSeed);
*)
   (* Call HandleRandom 9 times *)
  FOR Count := 1 TO 9 DO
    HandleRandom ( Count);
  WRITELN;
  WRITE ( 'Press ENTER to end program.');
  READLN ( Ch);
END.
```

This program generates nine random integers between 0 and 25000, inclusive. For every number generated, the program displays the value. In addition, for every third value, the program checks whether the value is even or odd, and tells you which it is.

The program illustrates two of CodeWarrior Pascal's control structures:

- The **IF** statement, which enables your program to do different things, depending on what conditions hold.
- The **FOR** statement, which enables your program to repeat an action or series of actions a specified number of times.

The program also shows an example of a procedure with a parameter. A *parameter* is a means by which you can pass information into and out of a procedure or function, in order to control how the routine (procedure or function) works on a particular call. You'll learn about parameters in Chapters 4 and 6.

In addition, program **2-2.PAS** gives you your first chance to add interface files to your project. Notice the listing line just under the program name: **USES Windows**. This line tells the compiler that the program will be using routines or data structures from units — files created to supplement the capabilities provided by the standard libraries.

You'll learn about units in Chapter 10. For now, take it on faith that this line makes it necessary for you to add six files into your project. These files are: **WINDOWS.P**, **TYPES.P**, **MACROS.P**, **VARIABLES.P**, **FUNCTIONS.P**, and **CONSTANTS.P**. The files are located in the **\CodeWarrior\Interfaces** folder. They are included in the program to give you access to the **RAND** random number generator and the **SRAND** random number seeder.

You can add these files to your project after you've created the rest of it. As you did in Chapter 1 and with the previous example in this chapter, open the stationery by starting a **New Project** using a **Win32x86** stationery and building a **Pascal Console Application**.

Add your source code file (**2-2.PAS**) and remove the placeholder source file (**HelloWorld.P**). At this point, your project will look similar to the one for **2-1.PAS**, except for the source file.

To add the interface files, you begin with the project as you've just created it.

- Click on the gray line just below your source file. This will let you add the interface files under the **Source** group. **2-2.PAS** should be highlighted after this step.
- Select the **Add Files** command from the **Project** menu.
- To specify the files you want, type **\CodeWarrior\Interfaces\*.p** in the **File Names** text box.

- Press ENTER. You'll get a list of all the files in the specified folder that have the extension **P**. There should be just the six files you'll need to add.
- To add them all at once, move to the first file in the list and click on it. The selected file will be highlighted.
- Then move to the last file on the list, and click on it *while holding down the Shift key at the same time*. All six files will now be highlighted.
- Click on the **Add** command in the dialog box. You'll see that the six files all have been added to the project.
- Get ready to build another program, which you'll do using the **Make** and **Run** commands.

If you should have other interface files in the **\CodeWarrior\Interfaces** directory, you can select multiple files individually by holding down the Control key while you click on each file you want to add. With this method, you'll have to select the six files one at a time, but you can then add them all at once.

As a little exercise, try to determine what parts of the program produced the different lines of the output. The comments in the program — text between (* and corresponding *) — should help you in this exercise.

Making the Program More Flexible

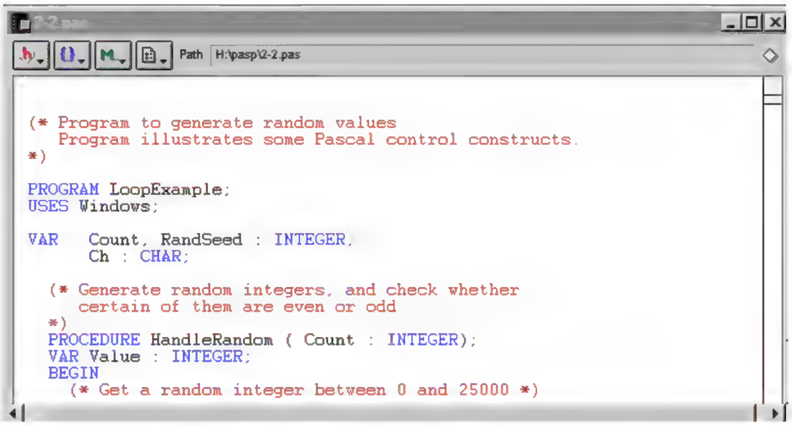
In its current version, the program will generate the same random sequence each time. This has to do with the way the random number generator works.

To get a different sequence you need to "seed" the random number generator. you can accomplish this in the program by

- Noting the red text in the five lines immediately after the **BEGIN** that starts the **Main program body** — beginning on the **(*** line and ending on the ***)** line.
- Deleting the **(*** immediately below the **BEGIN** for the **Main program body**.
- Deleting the ***)** immediately below the line containing the call to **SRAND** (which seeds the random number generator). This should reduce the five red lines to three black and blue lines. It should also give you a variable random number generator.
- Click on the close box to end the edit session, and click on **Save** to save your changes.
- Give the **Make** and **Run** commands to try out the new version of your program.

The Edit Window Revisited

While you're in the edit window making some changes, let's take a moment to examine some of the editor's features. Figure 2-9 shows the edit window with **2-2.PAS** as the work file. Notice the bar under the editor window's title bar. This bar contains four pop-up menus and path information about the file.



```
(* Program to generate random values
   Program illustrates some Pascal control constructs.
*)

PROGRAM LoopExample;
USES Windows;

VAR    Count, RandSeed : INTEGER,
       Ch : CHAR;

  (* Generate random integers, and check whether
     certain of them are even or odd
  *)
  PROCEDURE HandleRandom ( Count : INTEGER);
  VAR Value : INTEGER;
  BEGIN
    (* Get a random integer between 0 and 25000 *)
```

Figure 2.9 **The Edit window has four pop-up menus**

The leftmost pop-up menu (with the **.h** in the icon) is the *Interface Pop-Up Menu*. This menu displays a list of all the interface files used in your program. If you click on this icon, you'll see the six files you added.

The *Routine Pop-Up Menu* lists all the procedures and functions defined in your source file. The routine in which your current insert point is located is marked with a check. If you click on this menu while you're making your editing changes, you'll see that **LoopExample** (the main program) is checked — because your current location is in the main program.

The *Marker Pop-Up Menu* lets you mark specific locations in the file, so you can move easily to those locations. For example, you might want to mark the last place you were working, so that you can move there immediately when you get back to work.

The *Options Pop-Up Menu* lets you specify how your source file's text is to be displayed, and also the format in which the file is to be saved. In particular, this is where you can specify whether you want the editor to display different elements in your program (comments, reserved words, etc.) in different colors. If you click on this icon, you'll see that **Syntax Coloring** is checked.

Finally, if your edit window is open far enough, you'll see a box in the lower left corner. This box tells you the number of the line containing the current insertion point.

Example: Using Complex Data Types

Let's look at another example, which illustrates more features of Pascal, including the use of more complex and specialized data types, and the use of nested routines. *Nested routines* are defined inside another procedure or function, rather than in the declaration section of the main program.

The following program deals card hands to as many as four players. Each hand can have up to 10 cards. First, run the program a few times. Try dealing 5-card and 7-card hands. Vary the number of players being dealt cards. Note that this program also **USES Windows** — because it calls the random number generator and seeder.

Listing 2.3

```
(* Program to deal card hands to up to 4 players;
   program illustrates use of Array variables
   and of BOOLEAN and enumerated data types.
*)

PROGRAM DealDemo;
USES Windows;

CONST MaxHands = 4;      MaxCards = 12;
       (* Card values range from 2 through Ace (14) *)
      MinFace = 2;       MaxFace = 14;
TYPE SuitRange = ( club, diamond, heart, spade);
     FaceRange = MinFace .. MaxFace;
     (* Define an array containing a deck of cards.
        If a card is present, the corresponding cell
        value is TRUE.
     *)
     CardArray = ARRAY [ FaceRange, club .. spade ] OF BOOLEAN;
     (* Define a collection of decks,
        each corresponding to a player.
     *)
     Group = ARRAY [ 1 .. MaxHands] OF CardArray;
VAR Deck : CardArray;
    Players : Group;
    Ch : CHAR;
    NrCards, NrHands, Index, RandSeed : INTEGER;
```

```
 (* Initialize a deck to the specified value *)
PROCEDURE InitCards ( VAR Cards : CardArray;
                      Value : BOOLEAN);
VAR SuitCounter : SuitRange;
    FaceCounter : FaceRange;
BEGIN
  FOR SuitCounter := club TO spade DO
    FOR FaceCounter := MinFace TO MaxFace DO
      Cards [ FaceCounter, SuitCounter] := Value;
END; (* InitCards *)

 (* Display the contents of a particular deck *)
PROCEDURE DispCards ( VAR Cards : CardArray);
VAR SuitCounter : SuitRange;
    FaceCounter : FaceRange;

   (* Write a card number or J Q K or A *)
  PROCEDURE ShowCard;
  BEGIN
    IF FaceCounter < 11 THEN
      WRITE ( FaceCounter : 4)
    ELSE
    BEGIN
      CASE FaceCounter OF
        11 : WRITE ( 'J' : 4);
        12 : WRITE ( 'Q' : 4);
        13 : WRITE ( 'K' : 4);
        14 : WRITE ( 'A' : 4);
      END; (* CASE OF FaceCounter *)
    END;
  END; (* ShowCard *)

BEGIN  (* DispCards *)
  FOR SuitCounter := club TO spade DO
  BEGIN
    CASE SuitCounter OF
      club : WRITE ( 'clubs   : ');
      diamond : WRITE ( 'diamonds: ');
      heart : WRITE ( 'hearts  : ');
      spade : WRITE ( 'spades  : ');
    END; (* CASE OF SuitCounter *)
    FOR FaceCounter := MinFace TO MaxFace DO
      IF Cards [ FaceCounter, SuitCounter] = TRUE THEN
        ShowCard;
    WRITELN;
  END;
END; (* DispCards *)

(* Deal the specified # of cards to the players *)
PROCEDURE Deal ( VAR Deck : CardArray;
                 VAR Players : Group;
                 NrCards, NrHands : INTEGER);
VAR CurrSuit : SuitRange;
    CurrFace : FaceRange;
    CIndex, HIndex : INTEGER;

  (* Determine the "next" card in the deck *)
  PROCEDURE GetCard ( VAR TheFace : FaceRange;
                      VAR TheSuit : SuitRange);

    (* Determine the suit of the next card *)
    PROCEDURE GetSuit ( VAR TheSuit : SuitRange);
    VAR Result : INTEGER;
    BEGIN
      Result := RAND MOD 4;
      CASE Result OF
        0 : TheSuit := club;
        1 : TheSuit := diamond;
        2 : TheSuit := heart;
        3 : TheSuit := spade;
      END; (* CASE  OF Result *)
    END; (* GetSuit *)

  BEGIN  (* GetCard *)
    REPEAT
      GetSuit ( TheSuit);
      TheFace := RAND MOD ( MaxFace - MinFace + 1) + MinFace;
    (* Keep looking until a card is selected that is
       still in the deck.
      *)
      UNTIL Deck [ TheFace, TheSuit] = TRUE;
    END; (* GetCard *)

  BEGIN  (* Deal *)
    FOR CIndex := 1 TO NrCards DO
    BEGIN
      FOR HIndex := 1 TO NrHands DO
      BEGIN
        GetCard ( CurrFace, CurrSuit);
        Players [ HIndex, CurrFace, CurrSuit] := TRUE;
        Deck [ CurrFace, CurrSuit] := FALSE;
      END; (* FOR Hindex *)
    END; (* FOR CIndex *)
  END;  (* Deal *)

  (* Determine # players and # of cards per hand *)
  PROCEDURE GetGameInfo ( VAR NrCards, NrHands : INTEGER);
  BEGIN
    REPEAT
      WRITE ( '# of players? (Max = ', MaxHands, ') ');
      READLN ( NrHands);
    UNTIL NrHands <= MaxHands;
    REPEAT
      WRITE ( '# of Cards? (Max = ', MaxCards, ') ');
      READLN ( NrCards);
    UNTIL NrCards <= MaxCards;
  END; (* GetGameInfo *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN  (* Main program *)
  GetInt ( 'Integer value to seed the random number generator',
RandSeed);
  SRAND ( RandSeed);  (* "shuffle" deck generator *)
  GetGameInfo ( NrCards, NrHands);
  InitCards ( Deck, TRUE);
  FOR Index := 1 TO MaxHands DO
    InitCards ( Players [ Index], FALSE);

  Deal ( Deck, Players, NrCards, NrHands);

  (* Display the hands *)
  FOR Index := 1 TO NrHands DO
  BEGIN
    WRITELN ( 'PLAYER ', Index, ': ');
    DispCards ( Players [ Index]);
    WRITELN;
  END;
  (* 4 hands fill the screen, so wait before showing deck *)
  IF NrHands = 4 THEN
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END;
  (* then display the remainder of the deck. *)
  WRITELN ( 'DECK: ');
  DispCards ( Deck);
  WRITE ( 'Press ENTER to end program.');
  READLN (Ch);
END.
```

To understand the program, it helps to see how it is organized. The main program has three main sections:

- The *setup* section, which initializes important variables, and determines the constraints of the game (number of hands and players). This section consists of the first four statements, including the **FOR** statement.
- The *deal* section, which distributes the required number of cards to each player. The single call to **Deal** represents this section.
- The *display* section, which shows the contents of each player's hands and also shows the remaining cards in the deck. The remaining statements in the main program (ex-

cept the **READLN**, which keeps you on the output screen), make up the display section.

This program has a relatively short program body, but a rather long declaration section. Numerous procedures are defined, as well as some useful data types.

First, notice the **CONST** declarations near the top of the file. The first of these, for example, associates a meaningful name (**MaxHands**) with a special value in the program. This value is declared as a constant (rather than a variable) because it does not change throughout the program.

Giving the value a name makes it easier to change the actual numerical value if necessary. For example, suppose you needed to limit hands to five cards. You could simply change the value of **MaxHands** at the top of your file. No matter how often you refer to this value in your program, every reference by its name will automatically be modified the next time you compile the program.

In addition to these constants, the declaration section includes several type definitions, in which a type of information is described, by listing its possible values or by describing a template for an item of such information.

For example, the program defines a **SuitRange** type, which can take on **club**, **diamond**, **heart**, and **spade** as possible values. These values describe an *enumerated type*, in which you can use meaningful names to represent values, rather than having to use arbitrary numerical or letter codes to represent these values. You'll learn about enumerated types in Chapter 3.

FaceRange describes a type that can take on numerical values only within a certain range. These values are used as codes for card values. Values 2 through 10 are regular number cards, and higher values represent face cards (with 14 being an Ace).

The type definitions also include two templates. The first template describes a **CardArray** as a collection of every combination of four suits and 13 face values — that is, as something that could store information about a complete deck of cards.

The description not only shows that you can use arrays in Pascal, but also that you can use enumerated as well as numerical values to identify elements (cells) of an array. For example, you can refer to the cell corresponding to 7, club (which would be used to represent information about the seven of clubs).

In this program, the individual cells of such an array contain a value that can be either **TRUE** or **FALSE**. You'll learn about arrays in Chapter 7, and about **BOOLEAN**s (a data type that can take on only two possible values, **TRUE** and **FALSE**) in the next chapter.

The description of a **Group** also says it is an array. However, the individual elements of this array are actually **CardArrays**.

The next part of the program's declaration section shows how to declare variables of the types just described, as well as of other, predefined, types. This is followed by definitions for several procedures.

These procedures carry out the work requested in the main program, by carrying out the details involved in, for example, initializing a **CardArray** variable.

Procedure **GetGameInfo** determines how many players you want to have and how many cards each player will get. The procedure illustrates another Pascal control construct, the **REPEAT** loop. This loop is executed over and over until a certain condition becomes true.

The **InitCards** procedure initializes each cell in a **CardArray** variable to the specified initial value. For example, in the program, the **Deck** variable is initialized to **TRUE**. This means that, at the start of the program, you're playing with a full deck.

Each player's hand, on the other hand, is initialized to **FALSE**, since these hands start out empty. During the deal, cards will be selected at random from **Deck**, and will be assigned to a specified hand. As part of this process, the card is removed from the deck by setting its cell to **FALSE**. In contrast, the recipient's hand grows when a card is added. This is reflected in the fact that the appropriate cell is initialized to **TRUE**.

Procedure **InitCards** uses a nested **FOR** loop, in which the inner loop is executed once for each value the outer loop indicates. Thus, for each suit, the cards from the 2 through the Ace are initialized.

Procedure **Deal** is actually responsible for getting the cards to the players in the correct order. The source code for this procedure illustrates that procedures (and functions) can have other routines defined inside them. In Chapter 6 you'll learn how to define routines within other routines.

Deal does its work by first selecting a card that hasn't been dealt yet. This card is then added to the appropriate player's hand. This procedure illustrates *nested* **FOR** loops. you'll learn more about such techniques in Chapter 4.

The card selection is done by **GetCard**, which is nested in **Deal**. GetCard, in turn, uses **GetSuit** for part of its work.

Procedure **DispCard** checks each cell of a **CardArray**. If the cell contains **TRUE**, the card is present, so the routine needs to indicate this.

DispCard uses a nested routine, **ShowCards**, to indicate that a card is present. This nested procedure handles cards with face values of 11 through 14 (that is, face cards) differently.

Recycling Your Programs

It's possible to write Pascal programs in a way that makes it easy to adapt them for other purposes. To illustrate this, and also to introduce a few more IDE commands, let's create a new program — based on file **2-3.PAS**.

The new program is designed to deal hands using a pinochle deck. Pinochle is played with a 48-card deck. Only face values from 9 through Ace are used, and each card appears twice in the deck. Thus, there are two Queens of Hearts, two 10s of clubs, and so forth.

As you'll see, there are very few changes you need to make to the **DealDemo** program. Let's create this new program in a file called **2-4.PAS**.

To simplify your task, you can simply copy the contents of **2-3.PAS** to **2-4.PAS**, and then just edit the new file. To do this,

- Open file **2-3.PAS** in the editor.
- Select the **Select All** command from the **Edit** menu (or use the Ctrl+A shortcut). This will highlight all the text in the file.
- Select the **Copy** command from the **Edit** menu (or use the Ctrl+C shortcut). This will put a copy of the file in the Clipboard.
- Select the **New** command from the **File** menu. This will put you in the edit window for a new file (currently named **Untitled**).
- Select **Paste** from the **Edit** menu (or use the Ctrl+V shortcut). This will copy the contents of the Clipboard — that is, **2-3.PAS** — to your work file.

The first two changes are near the top of the file, so you can easily use the arrow keys to reach them. Change the value of **MaxCards** to 12, and of **MinFace** to 9. (Actually, there are pinochle games in which players get more than 12 cards, but we're keeping things simple here.)

Instead of storing true or false values in the cells, this program needs to store how many of a particular card are in the deck or hand. This value can be 0, 1, or 2. In the program, this will mean changing **BOOLEAN** to **INTEGER** in the definition of the **CardArray** type.

You can use CodeWarrior Pascal's Find and Replace command to accomplish this. Select **Find** from the **Search** menu (or type CTRL+F to activate the find and replace command). In the dialog box, you can specify both the old (find) and new (replace) text. Type **BOOLEAN** in the **Find** box, then click in the **Replace** box. In the **Replace** box, type **INTEGER**. Then click on the **Find** command.

When the editor finds the first occurrence of the **Find** string, you can decide how to handle it. Usually the action will be one of **Replace All**, Replace, or **Replace & Find Next**. At the bottom of the **Find** dialog box, you can see several check boxes. These allow you to fine-tune the search. See the *User's Guide* for more information on these options.

There are three places where **BOOLEAN** occurs. One of these is in a comment. For simplicity, find and replace each of these.

Since the erstwhile **BOOLEAN** variables are now integers, you also need to make changes in any **BOOLEAN** values. That is, you'll need to change every occurrence of **TRUE** and **FALSE** to a numerical value. **FALSE** will become zero, since this is the numerical equivalent of not having a card. **TRUE** may become 1 or 2, or it may be changed to a different comparison. For example, one of the lines you'll find when you search for **TRUE** is the following:

```
UNTIL Deck [ TheFace, TheSuit] = TRUE;
```

Change the **= TRUE** to **> 0**.

Use whatever means you feel most comfortable with to move to the two assignment statements in the center of **Deal**, and change these to the following:

```
INC ( Players [ HIndex, CurrFace, CurrSuit]);
DEC ( Deck [ CurrFace, CurrSuit]);
```

The first statement increases the value representing a particular card. This means that the player has been dealt another of these cards. The second statement removes that card from the deck, by decreasing the value in the cell corresponding to the card.

You need to add a new variable in procedure **DispCards**. Move the cursor to the top of the file, and then search for

```
PROCEDURE DispCards
```

Once you're on this line, move the cursor to the line under the one that begins with **FaceCounter**. Add the following line:

```
Counter : INTEGER;
```

Then move to the **IF** statement near the end of this procedure. Replace the **IF** line with the following:

```
FOR Counter := 1 TO Cards [ FaceCounter, SuitCounter] DO
```

There are just two minor changes in the main program body. In the calls to **InitCards**, replace the **TRUE** with 2 in the first call, and the **FALSE** with 0 in the second call. These substitutions reflect the fact that a pinochle deck begins with two copies of each card, and a pinochle hand begins with 0 cards.

All right, you're ready to run your program. The complete program (with minimal changes in the comments) is shown in the following listing. (You should update the comments as well. First, it will make things less confusing if you should ever look at this program at some future time. Second, it will force you to look at the program, which can often yield insight.

Listing 2.4

```
(* Program to deal pinochle hands to up to 4 players;
   program illustrates use of Array variables
   and of INTEGER and enumerated data types.
*)
```

```
PROGRAM PinochDemo;
USES Windows;

CONST MaxHands = 4;      MaxCards = 12;
       (* Card values range from 2 through Ace (14) *)
      MinFace = 9;       MaxFace = 14;
TYPE SuitRange = ( club, diamond, heart, spade);
     FaceRange = MinFace .. MaxFace;
     (* Define an array containing a deck of cards.
        If a card is present, the corresponding cell
        value is TRUE.
     *)
     CardArray = ARRAY [ FaceRange, club .. spade ] OF INTEGER;
     (* Define a collection of decks,
        each corresponding to a player.
     *)
     Group = ARRAY [ 1 .. MaxHands] OF CardArray;
VAR Deck : CardArray;
    Players : Group;
    Ch : CHAR;
    NrCards, NrHands, Index, RandSeed : INTEGER;

   (* Initialize a deck to the specified value *)
  PROCEDURE InitCards ( VAR Cards : CardArray;
                        Value : INTEGER);
  VAR SuitCounter : SuitRange;
      FaceCounter : FaceRange;
  BEGIN
    FOR SuitCounter := club TO spade DO
      FOR FaceCounter := MinFace TO MaxFace DO
        Cards [ FaceCounter, SuitCounter] := Value;
  END; (* InitCards *)

   (* Display the contents of a particular deck *)
  PROCEDURE DispCards ( VAR Cards : CardArray);
  VAR SuitCounter : SuitRange;
      FaceCounter : FaceRange;
      Counter : INTEGER;
  (* Write a card number or J Q K or A *)
  PROCEDURE ShowCard;
  BEGIN
    IF FaceCounter < 11 THEN
      WRITE ( FaceCounter : 4)
    ELSE
    BEGIN
      CASE FaceCounter OF
        11 : WRITE ( 'J' : 4);
        12 : WRITE ( 'Q' : 4);
        13 : WRITE ( 'K' : 4);
        14 : WRITE ( 'A' : 4);
      END; (* CASE OF FaceCounter *)
    END;
  END; (* ShowCard *)

BEGIN  (* DispCards *)
  FOR SuitCounter := club TO spade DO
  BEGIN
    CASE SuitCounter OF
      club : WRITE ( 'clubs   : ');
      diamond : WRITE ( 'diamonds: ');
      heart : WRITE ( 'hearts  : ');
      spade : WRITE ( 'spades  : ');
    END; (* CASE OF SuitCounter *)
    FOR FaceCounter := MinFace TO MaxFace DO
      FOR Counter := 1 TO Cards [ FaceCounter, SuitCounter] DO
        ShowCard;
    WRITELN;
  END;
END; (* DispCards *)

(* Deal the specified # of cards to the players *)
PROCEDURE Deal ( VAR Deck : CardArray;
                 VAR Players : Group;
                 NrCards, NrHands : INTEGER);
VAR CurrSuit : SuitRange;
    CurrFace : FaceRange;
    CIndex, HIndex : INTEGER;
    (* Determine the "next" card in the deck *)
    PROCEDURE GetCard ( VAR TheFace : FaceRange;
                        VAR TheSuit : SuitRange);

      (* Determine the suit of the next card *)
      PROCEDURE GetSuit ( VAR TheSuit : SuitRange);
      VAR Result : INTEGER;
      BEGIN
        Result := RAND MOD 4;
        CASE Result OF
          0 : TheSuit := club;
          1 : TheSuit := diamond;
          2 : TheSuit := heart;
          3 : TheSuit := spade;
        END; (* CASE  OF Result *)
      END; (* GetSuit *)

    BEGIN  (* GetCard *)
      REPEAT
        GetSuit ( TheSuit);
        TheFace := RAND MOD ( MaxFace - MinFace + 1) + MinFace;
      (* Keep looking until a card is selected that is
         still in the deck.
      *)
      UNTIL Deck [ TheFace, TheSuit] > 0;
    END; (* GetCard *)

  BEGIN  (* Deal *)
    FOR CIndex := 1 TO NrCards DO
    BEGIN
      FOR HIndex := 1 TO NrHands DO
      BEGIN
        GetCard ( CurrFace, CurrSuit);
        INC ( Players [ HIndex, CurrFace, CurrSuit]);
        DEC ( Deck [ CurrFace, CurrSuit]);
      END; (* FOR Hindex *)
    END; (* FOR CIndex *)
  END;  (* Deal *)

  (* Determine # players and # of cards per hand *)
  PROCEDURE GetGameInfo ( VAR NrCards, NrHands : INTEGER);
  BEGIN
    REPEAT
      WRITE ( '# of players? (Max = ', MaxHands, ') ');
      READLN ( NrHands);
    UNTIL NrHands <= MaxHands;
    REPEAT
      WRITE ( '# of Cards? (Max = ', MaxCards, ') ');
      READLN ( NrCards);
    UNTIL NrCards <= MaxCards;
  END; (* GetGameInfo *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN  (* Main program *)
  GetInt ( 'Integer value to seed random number generator',
RandSeed);
  SRAND ( RandSeed);  (* "shuffle" deck generator *)
  GetGameInfo ( NrCards, NrHands);
  InitCards ( Deck, 2);
  FOR Index := 1 TO MaxHands DO
    InitCards ( Players [ Index], 0);

  Deal ( Deck, Players, NrCards, NrHands);

  (* Display the hands *)
  FOR Index := 1 TO NrHands DO
  BEGIN
    WRITELN ( 'PLAYER ', Index, ': ');
    DispCards ( Players [ Index]);
    WRITELN;
  END;
  (* 4 hands fill the screen, so wait before showing deck *)
  IF NrHands = 4 THEN
   READLN ( Ch);
  (* then display the remainder of the deck. *)
  WRITELN ( 'DECK: ');
  DispCards ( Deck);
  WRITELN ( 'Press ENTER to end program');
  READLN ( Ch);
END.
```

Now that you've managed that conversion, think about what you would need to do to change either dealing program to work for blackjack. Perhaps when you've learned more about the language features used in the program (arrays, control structures, procedures and functions) you'll want to build a blackjack program.

Example: Dealing with Files

Let's look at another example, to illustrate some of the more advanced features of Pascal — such as files. Program **Bubble** (2-5.PAS) does the following:

- Reads up to 200 (**MaxSize**) lines from a specified file
- Sorts these lines
- Writes the sorted lines to another file

Listing 2.5

```
(* Program to sort lines read from a file,
   and to write the sorted lines to another file.
*)

PROGRAM Bubble;

CONST MaxSize = 200;
TYPE  String80Array = ARRAY [ 1 .. MaxSize] OF STRING [ 80];
VAR   Str : String80Array;
      NrVals, Count : INTEGER;
```

```
    Source, Log : Text;
    Ch : CHAR;
    SourceOK, LogOK : BOOLEAN;
    SourceName, LogName : STRING;

PROCEDURE GetString ( Message : STRING;
                      VAR Info : STRING);
BEGIN
  WRITE ( Message, '? ');
  READLN ( Info);
END; (* GetString *)

(* Sort an array of strings by letting the "largest"
   remaining string work its way to the top of an
   array on each pass through the array.
*)
PROCEDURE BubbleSort ( VAR Info : String80Array;
                           Size : INTEGER);
VAR Temp : STRING [ 80];
    Low, High, Top : INTEGER;
BEGIN
  Top := Size + 1;
  WHILE ( Top > 1) DO
  BEGIN
    Low := 1;
    High := 2;
    WHILE High < Top DO
    BEGIN
      (* if lower string needs to move upward in the array *)
      IF Info [ Low] > Info [ High] THEN
      BEGIN
        Temp := Info [ Low];
        Info [ Low] := Info [ High];
        Info [ High] := Temp;
      END; (* If lower string needs to move upward *)
      Low := Low + 1;
      High := High + 1;
    END; (* WHILE High < Top *)
    Top := Top - 1;
  END; (* WHILE Top > 1 *)
  END;  (* BubbleSort *)

  (* Get file names, if necessary.
     Open the source file, and create a log file.
  *)
  PROCEDURE HandleFiles;
  BEGIN
    GetString ( 'Source file name', SourceName);
    GetString ( 'Log file name', LogName);
    WRITELN ( 'Source name = ', SourceName);
    WRITELN ( 'Log name    = ', LogName);
    (*  ASSIGN ( Source, SourceName);  *)
    (*$I-*)
    (* RESET ( Source);   *)
    RESET ( Source, SourceName);
    (*$I+*)
    IF IOResult = 0 THEN
      SourceOK := TRUE
    ELSE
      SourceOK := FALSE;
    (*  ASSIGN ( Log, LogName);  *)
    (*$I-*)
    (*  REWRITE ( Log);  *)
    REWRITE ( Log, LogName);
    (*$I+*)
    IF IOResult = 0 THEN
      LogOK := TRUE
    ELSE
      LogOK := FALSE;
  END; (* HandleFiles *)

  PROCEDURE GetLines;
  BEGIN
    (* Get the lines to be sorted *)
    Count := 0;
    WRITELN ( 'dbTMP: In GetLines, before loop, Count = ',
Count);
    WHILE ( ( NOT ( EOF ( Source))) AND ( Count < MaxSize)) DO
    BEGIN
      Count := Count + 1;
      READLN ( Source, Str [ Count]);
      WRITELN ( 'dbTMP: ', Str[ Count]);
    END;
    NrVals := Count;
    WRITELN ( 'dbTMP: In GetLines, after loop, Count = ', Count);
  END; (* GetLines *)

BEGIN  (* Main Program *)
  HandleFiles;

  IF SourceOK AND LogOK THEN
  BEGIN
    GetLines;

    BubbleSort ( Str, NrVals);   (* sort the lines *)

    (* Write the sorted lines to the log file *)
    FOR Count := 1 TO NrVals DO
    BEGIN
      WRITELN ( Str [ Count]);
      WRITELN ( Log, Str [ Count]);
    END;
    WRITELN ( 'Log file written. Press ENTER to end program');
    CLOSE ( Source);
    CLOSE ( Log);
  END
  ELSE
    WRITELN ( 'File Error. Press ENTER to end program');
  READLN ( Ch);
END.
```

Let's look at some of this program's elements and features. The fundamental data type is an array of strings, described in the following line from the program:

```
TYPE  String80Array = ARRAY [ 1 .. MaxSize] OF STRING [ 80];
```

This line describes a 200-element array. Each element in this array is a string of up to 80 characters. The string length limitation is indicated by the **[80]** at the end of the line. Recall that earlier programs used variables of type **STRING**. Such variables could be up to 255 characters long.

Since lines from text files are rarely longer than 80 characters, you can save a great deal of space by working with the 80-character strings. You'll see lots more uses for such strings throughout the book.

The two files used in the program are declared as variables of type **Text**. Before you can use these files you must open or create them. This is done in procedure **HandleFiles**. The listing actually includes code for two different ways of handling files, with one variant being commented out. In its current form, the program *does not* use Turbo Pascal I/O; rather, it uses the default method for CodeWarrior Pascal. In Chapter 8, you'll learn the difference, and you'll create programs using Turbo Pascal I/O as well as the default method. (The syntactical difference has to do with the way in which a file variable and a file name are associated with each other.)

File **Source** must already exist, so the program can read from it. For this reason, you just need to open the file; **Log**, on the other hand, is created anew each time. Any old log files with the same name are lost. You'll learn more about the syntax of file opening and creation in Chapter 8.

After the files have been prepared in procedure **HandleFiles**, the program reads the lines from **Source**. This takes place within the **WHILE** loop in procedure **GetLines**.

The **WHILE** loop executes as long as the end of the source file has not been reached and as long as fewer than 200 lines have been read. If the file contains fewer than 200 lines, then the end of the file will eventually be reached, and the loop will stop.

After sorting the array, the lines are written to the log file. The last part of the program closes the files — to save their contents.

The Bubblesort Algorithm

The core of the program is actually the single statement in which **BubbleSort** is called. This procedure sorts the array of strings using what is known as a *bubblesort*, or *exchange*, algorithm.

The sorting is based on the ASCII code of the letters with which each line begins. This is roughly an alphabetical ordering, except that all lines that begin with an uppercase letter will precede those that begin with a lower case letter.

Thus, a line that begins with `A' would be near the front of the sorted array, and one that begins with `z' would be near the end. Because of the way the ASCII character set is arranged, lines that begin with `Z' would precede those that begin with `a.' In the discussion, we'll refer to strings that come later in a sorted list as “larger" strings.

The basic strategy behind the bubblesort algorithm is as follows:

- Make repeated passes through the array.

- On each pass, compare successive cells. For example, on the first pass the algorithm would compare cells 1 and 2, then cells 2 and 3, 3 and 4, and so forth, until all the remaining cells have been compared.
- If necessary, exchange the strings in two cells — so that the "larger" string ends up in the higher cell.
- When all the cell pairs have been compared, the top cell will contain the string that comes last in the sorting. Thus, after the first pass through the array, cell 200 will contain the string that comes last in the sorted list. (The highest cell may have an index less than 200 if fewer than 200 lines were read.)
- The next time through the array, you need not look at the last cell, since you know that cell's string is in its proper place. That means you only need to examine 199 cells the second time through.
- After this pass, the 199th cell will contain the last line from the 199 remaining lines.
- Continue this process until there are no more passes to make.

You'll get another chance to work with the bubblesort algorithm in Chapter 7.

Two Final Examples

Let's look at two more examples, to give you the chance to refresh and reinforce your mastery of the CodeWarrior Pascal editor. The two programs will also give you a chance to test your recall of the project creation commands.

The following program (**2-6.PAS**) lets you determine the maximum of a set of values. To provide the values, simply enter them.

When you've finished entering your values, enter 0.0 as your last value. This value tells the program that you're finished. This last value *must* be included at the end of your data entry, but is not included in any counts or computations.

Listing 2.6

```
(* Program to compute the maximum of a set of values. *)

PROGRAM GetMaxValue;

CONST StopValue = 0.0;     (* value to indicate end of input *)
VAR CurrMax, CurrVal : REAL;
```

```
    Count : INTEGER;
    Ch : CHAR;

  (* Return the larger of two values passed as parameters *)
  FUNCTION FindMax ( Val1, Val2 : REAL) : REAL;
  BEGIN
    IF Val1 > Val2 THEN
      FindMax := Val1
    ELSE
      FindMax := Val2;
  END; (* FindMax *)

BEGIN
  Count := 0;
  CurrMax := -1000000.0;

  WRITELN ( 'Enter values, with ', StopValue: 3 : 1,
            ' as the last value.');
  (* Read values until the StopValue is encountered *)
  READ ( CurrVal);
  WHILE CurrVal <> StopValue DO
  BEGIN
    Count := Count + 1;
    CurrMax := FindMax ( CurrMax, CurrVal);
    READ ( CurrVal);
  END; (* WHILE NOT StopValue *)

  (* Display results *)
  WRITELN ( Count, ' values read: Max = ', CurrMax : 10 : 3);
  WRITE ( 'Press ENTER to end program.');
  READLN ( Ch);
END.
```

Now, use the editing commands (such as Find and Replace) to modify the program to find the *minimum* of a set of values. When you're done, the program will look like the one in the following listing, which shows file **2-7.PAS**.

Listing 2.7

```
(* Program to compute the minimum of a set of values. *)

PROGRAM GetMinValue;

CONST StopValue = 0.0;    (* value to indicate end of input *)
VAR CurrMin, CurrVal : REAL;
    Count : INTEGER;
    Ch : CHAR;

  (* Return the larger of two values passed as parameters *)
  FUNCTION FindMin ( Val1, Val2 : REAL) : REAL;
  BEGIN
    IF Val1 < Val2 THEN
      FindMin := Val1
    ELSE
      FindMin := Val2;
  END; (* FindMin *)

BEGIN
  Count := 0;
  CurrMin := 1000000.0;

  WRITELN ( 'Enter values, with ', StopValue: 3 : 1,
            ' as the last value.');
  (* Read values until the StopValue is encountered *)
  READLN ( CurrVal);
  WHILE CurrVal <> StopValue DO
  BEGIN
    Count := Count + 1;
    CurrMin := FindMin ( CurrMin, CurrVal);
    READLN ( CurrVal);
  END; (* WHILE NOT StopValue *)

  (* Display results *)
  WRITELN ( Count, ' values read: Min = ', CurrMin : 10 : 3);
  WRITE ( 'Press ENTER to end program.');
  READLN ( Ch);
END.
```

Summary

In this chapter, you got an informal introduction to Pascal, and you had the opportunity to use the CodeWarrior Pascal environment. In the next chapter, you'll start getting a more systematic introduction to procedures and functions in Pascal.

3

CodeWarrior Pascal's Simple Data Types

Pascal includes a rich variety of data types, to make it easier for you to represent different kinds of information in the most useful way. Pascal also enables you to create your own data types.

A *data type* provides a template within which specific information will be stored. For example, an **INTEGER** is a data type that will be used to store whole numbers, and that will be allocated two bytes of storage. Similarly, an **ARRAY** is a data type that will hold multiple values, all having the same form. The storage allocated for such a type depends on the number and types of elements. For example, an array of 100 **INTEGER**s would be allocated 200 bytes, and could be used to store 100 whole numbers.

CodeWarrior Pascal extends the standard version of Pascal. It provides an even wider collection of data types, and makes the language more flexible and easier to use. A variable's type determines the amount of storage allocated for the variable and the kinds of operations you can carry out with the variable. Both of these are important for the compiler.

Pascal provides data types to represent both simple and complex information. This chapter will present some of the simple types that CodeWarrior Pascal makes available to the programmer.

Before looking at data types, however, it will be useful to discuss some more general issues relating to the components of Pascal programs and to the way programs are put together.

Throughout this book, you'll find various suggestions about indentation, naming, and comments in your programs. You may want to do things differently than they're done here. The important thing is to develop your own programming style and to use it consistently.

Pascal Programs

Statements are the basic elements of a Pascal program. Essentially, programs are executed statement by statement. Statements represent the individual instructions required to complete the task the program is designed to do. For example, you'll use assignment statements to store information in program variables. You'll also have statements that call procedures and functions to compute values and to do the required work.

You have considerable leeway in how you write your statements. You can leave as much space as you wish between elements of a statement; you can even put different elements on different lines. Just about the only thing you can't do when writing the source code is to break in the middle of a word or a string. For example, the following listing shows three versions of the same statement:

```
MyInt := 7      (* version 1 *)

MyInt:=7        (* version 2 *)

MyInt
:=
7               (* version 3 *)
```

Notice that a statement can be split over multiple lines, as in the third version.

On the other hand, the next version of the statement is invalid, since there is a space in the middle of a variable name.

```
My Int := 7    (* invalid statement *)
```

This flexibility can be useful, since it allows you to use spacing and indentation to help organize your program. However, when processing your source file, the compiler needs to know where one statement ends and another begins. To provide this information, Pascal use semicolons (;) to separate statements. Because semicolons separate (as opposed to terminating) statements, you don't need a semicolon after the last statement of a program, procedure, or function; however, you do need a semicolon if another statement follows in the program, procedure, or function.

The following listing includes three places where a semicolon is optional:

Listing 3.1

```
(* Program to illustrate use, and non-use, of semicolons *)

PROGRAM SemiColons;
```

```
VAR IntVal : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch)          (* ; can be omitted here, *)
                          (*   since no statements follow *)
  END; (* AwaitUser *)

   (* Gets an integer from the procedure. *)
  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
   WRITE ( Message, ' ');
   READLN ( Value)        (* ; can be omitted here, *)
  END;

BEGIN
  GetInteger ( 'Value?', IntVal);
  WRITELN ( IntVal);
  AwaitUser               (* ; can be omitted here *)
END.
```

In this example, the semicolon is omitted before each of the **END**s — the ones that terminate **AwaitUser**, **GetInteger** and the main program. A semicolon is also optional after the last statement in a compound statement. (Recall that compound statements are bounded by **BEGIN** and **END**.) That is, you can omit the semicolon before the **END** that terminates the compound statement. You may, of course, include a semicolon in any of these places. This book includes the optional semicolons for consistency.

There are also a few places where you *cannot* include a semicolon. These places are discussed as relevant.

Notice that you also use a semicolon after the procedure heading and after each variable declaration line.

TIP: In Pascal, semicolons are used to separate statements.

Comments

You should include notes and comments to yourself in your source code file. This information can help you if you need to change or fix the program.

In Pascal, there are two ways of specifying comments: You can put the material between (* and *) or between { and }. When the compiler sees a (* or a { it ignores any text from that point until it reaches the corresponding *) or }.

WARNING! Be careful not to comment out lines of code accidentally.

The following program shows a mistake that can get you into trouble if you're not careful. Run the program to see what the output is. (Actually, the CodeWarrior Pascal IDE helps prevent this kind of error by allowing you to have the editor display different program elements — such as comments — in different colors.)

Listing 3.2

```
(* Program illustrating difficulties that can arise
   through incorrect use of comments.
   The middle line will not be displayed.
*)

PROGRAM CommentError;
VAR Ch : CHAR;

BEGIN  (* Main Program *)
  WRITELN ( 'One');  (* Start a comment
  WRITELN ( 'Two');     finish a comment *)
  WRITELN ( 'Three');
  WRITE ( 'Press ENTER to continue');
  READLN ( Ch);
END.
```

The comment in this program starts after the first call to **WRITELN**, and ends on the next line.

In such a case, the intent may be to provide comments for the first and second statements. However, the compiler reads the entire second line of the main program as part of the comment. As a result, the output from the program displays **One** and **Three**, but not **Two**.

The following listing shows the correct way to include comments for the first two lines. Compare the output when you run this program with the output from the **CommentError** program.

Listing 3.3

```
(* Program illustrating how to fix a faulty comment *)

PROGRAM CommentFix;
VAR Ch : CHAR;

BEGIN
  WRITELN ( 'One');   (* Start and finish a comment *)
  WRITELN ( 'Two');   (* start and finish another comment *)
  WRITELN ( 'Three');
  WRITELN ( 'Press ENTER to continue.');
  READLN ( Ch);
END.
```

You can include comments anywhere in your program, except in the middle of a name (such as a procedure or variable name). You should get into the habit of documenting your programs by including generous and clear comments.

You cannot nest comments — that is, you can't have a comment inside another comment. For example, the following line would lead to an error:

```
(* You cannot have (* nested *) comments *)
```

While the compiler is moving through a comment, it simply checks for the appropriate end of comment symbols. That is, if the comment begins with (*, the compiler looks for a matching *); if the comment begins with {, the compiler looks for a matching }.

As soon as it finds this symbol, the compiler assumes that the comment is finished and resumes processing your program file. In the preceding listing, the compiler would process

```
comments *)
```

as part of the source file, causing a compiler error.

Actually, you can fool the compiler into accepting nested comments because you can use two different symbols around comments. To put one comment inside another — should you ever find a good reason for doing so — use one set of symbols for the outer comment and the other set for the nested comment. The following listing shows nested comments that the compiler would accept:

```
(* You can have { quasi-nested } comments *)
```

Variables

In your programs, you'll want to save various items of information. Since many of these items will change their values during the course of program execution, your program needs to provide some way of saving these values.

A
variable is a named area of memory that is used to store such information. When writing a program, you set aside memory by declaring variables. When you declare a variable, you specify a name and a type for it. The compiler finds an available area of memory and associates the location of this memory with the name that you've provided.

The location of this memory is known only to the program; you needn't concern yourself with this. To read the information stored at that location or to write something there, you simply need to specify the variable name in the appropriate context. You'll learn how do this later in the chapter.

Naming Variables

When programming, you'll declare and use named constants and variables to represent particular values. You'll also name and define procedures and functions to carry out specific tasks. This section describes what constitutes a name in Pascal.

In CodeWarrior Pascal, a *name*, or *identifier*, must begin with a letter (*a* through *z* or *A* through *Z*). This letter may be followed by any number of letters, digits (*0* through *9*), and underscores (_). (Pascal, as Wirth defined it, does not allow underscores in identifiers, but most of the common Pascal implementations do. If you plan to compile your programs in environments other than CodeWarrior Pascal, don't use underscores in your identifiers, just to be on the safe side.)

The identifier can be as long as you want. The compiler, however, will read only the first 255 characters.

The first seven identifiers in the following list are valid in CodeWarrior Pascal; the remaining ones are not:

```
(* valid *)
AValidName
AlsoVALID
Likewise3
Tea4Two
Q
HARDTOREADBUTVALID
Valid_in_CodeWarriorPascal    (* not valid in standard Pascal
*)

(* invalid *)
Not-Valid                       (* dash (-) is not allowed *)
3Likewise                     (* first character is a digit *)
Split Name                   (* illegal space between words *)
```

In Pascal, the case (upper or lower) of the characters in a name is *not* significant; all characters are treated as if they were the same case. Thus, the following three names are all considered equivalent:

```
Version1
VERSION1
VeRsIoN1
```

The CodeWarrior Pascal environment has a Project Settings option that lets you tell the comnpiler to
treat variable names as case-sensitive. You can find and set this option on the Pascal Compiler tag on the
Prohect Settings dialog box.

TIP: In CodeWarrior Pascal, valid identifiers begin with a letter. This letter may be followed by letters, digits, and underscores. The case of the letters does not matter.

Restrictions on Names

There are certain words that you can't use as identifiers, since these are *reserved keywords* of the Pascal language. For example, you can't use the word **PROCEDURE** as an identifier for a variable, because this word specifies the beginning of a procedure definition. Similarly, the word **IF** is reserved for use in constructs used to make decisions in your program.

The following list contains the reserved keywords in standard Pascal:

AND
ARRAY
BEGIN
CASE
CONST
DIV
DO
DOWNTO
ELSE
END
FILE
FOR
FORWARD
FUNCTION
GOTO
IF
IN
LABEL
MOD
NIL
NOT
OF
OR
PACKED
PROCEDURE
PROGRAM
RECORD
REPEAT
SET
THEN
TO
TYPE
UNTIL
VAR
WHILE
WITH

You've seen several of these words in the example programs in Chapter 2. CodeWarrior Pascal has several additional reserved keywords, which are shown in the following list:

ASM
C
CYCLE
EXTERNAL
FAR
IMPLEMENTATION
INHERIT
INLINE
INTERFACE
LEAVE
OBJECT
OTHERWISE
OVERRIDE
STRING
UNIT
UNIV
USES

Pascal also has some keywords that are *predefined identifiers* but are not reserved. This means that you can use these words as identifiers; however, it's a very bad idea to do so. For example, consider the potential confusion if you used the word **INTEGER** as an identifier.

The following list shows standard Pascal's predefined identifiers:

ABS
ARCTAN
BOOLEAN
CHAR
FALSE
GET
INPUT
INTEGER
PACK
PRED
PUT
READ
SQR
SQRT
SUCC
TEXT

CHR
COS
DISPOSE
EOF
EOLN
EXP
LN
MAXINT
NEW
ODD
ORD
OUTPUT
READLN
REAL
RESET
REWRITE
ROUND
SIN
TRUE
TRUNC
UNPACK
WRITE
WRITELN

Some of these identifiers (such as **REAL**, **CHAR**, and **INTEGER**) represent data types. Others (such as **FALSE**, **TRUE**, and **MAXINT**) represent predefined values. Still others (such as **ABS**, **WRITELN**, **READ**, and **NEW**) represent predefined procedures and functions. You'll learn about the entities that correspond to these identifiers at the appropriate places in the book.

WARNING! It is very bad programming practice to use predefined identifiers as names for your variables.

You may use whatever identifiers you want in your programs. However, you should use meaningful names. Your program will be much easier to read (especially six months from now) if you use names such as **RunningTotal**, rather than **X**, for your variables.

In this book, we'll use only letters and digits in identifiers. This makes it easier to transport the programs for use in other environments. To distinguish individual words in an identifier, the first letter of each word will be in uppercase — for example, **ASampleNameWithMultipleWords** instead of **ASAMPLENAMEWITHMULTIPLEWORDS**, **asamplenamewithmultiplewords**, or **a_sample_name_with_multiple_words**.

Reserved words and predefined identifiers will be written in all uppercase.

Variable Declarations

Pascal is an example of a *strongly-typed language*. This means that the compiler must know the type of information being stored in a variable slot before the variable can be used. Furthermore, the compiler will enforce restrictions on the operations allowed with different types.

In order to specify the type of a variable, you must *declare* the variable. You've seen this in the example programs of Chapter 2. In a variable declaration, you specify a variable name and a type for the variable. You can have multiple variable declarations. The first line of declarations begins with the word **VAR**. Subsequent lines begin with identifiers or whitespace (blanks or tabs). The name and type information are separated by a colon (:).

For example, the following lines appeared in a program in Chapter 2:

```
VAR CurrMax, CurrVal : REAL;
    Count : INTEGER;
```

These two lines declare three variables. The first line declares two variables of type **REAL**. (You'll find information about this type later in this chapter.) If you declare two variables of the same type, you can list them together if you separate the individual names by commas.

The second line declares **Count** to be a variable of type **INTEGER**. You'll also learn about integers in greater detail later in the chapter. The indentation is to make the source code easier to read.

The material within the angle brackets (< >) specifies the sort of value to include at that point. Material within curly braces ({ }) is optional, but may be repeated multiple times. Material within double quotes must appear exactly as stated. You'll generally include a semicolon after the declaration — to separate it from the next line.

You can leave as many spaces as you wish between variable names, but you cannot leave a space within a variable name. You can use multiple lines to list your variable declarations. For example, the following declarations are all valid ways to specify variables:

```
(* a variable list can be split over multiple lines *)
Val1, Val2, Val3,
Val4, Val5, Val6 : REAL;

(* each variable also can be declared separately *)
IVal1 : INTEGER;
IVal2 : INTEGER;
IVal3 : INTEGER;
IVal4 : INTEGER;
IVal5 : INTEGER;
```

When you declare a variable, the compiler allocates a memory location that will be used to store the variable's value. The amount of storage allocated depends on the variable's type.

Pascal does *not* initialize variables when you declare them. For this reason, you can't make any assumptions about the starting contents of program variables — unless you assign them values in the program.

Failure to initialize a variable can lead to unpredictable results, as in the following program. Run the program in different sessions (for example, when you've been doing different things on the computer), to see what the output is.

Listing 3.4

```
(* Program to illustrate what can happen if you use
   an uninitialized variable.
   NOTE: In CodeWarrior Pascal, this program actually
         produces the same output each time.
         THIS IS NOT GUARANTEED, however.
*)

PROGRAM NoInitials;

VAR Value : INTEGER;
    Ch : CHAR;

BEGIN
   (* Value is uninitialized in the program *)
  Value := Value + 5;
  WRITELN ( Value);
  WRITE ( 'Press ENTER to continue ');
  READLN ( Ch);
END.
```

The output from this program will depend on whatever bit pattern was stored in the memory allocated for **Value** when the program starts. (Actually, in CodeWarrior Pascal, the outcome for this particular program will always yield 5. However, you *cannot* count on this to be the case in all programs. *Initialize your variables*.)

WARNING! Standard Pascal does not initialize program variables. It's up to you to make sure that variables are initialized before use.

CodeWarrior Pascal, however, *does* provide a way to initialize variables explicitly. You can give a variable an initial value in the **CONSTANT** declaration section, as described in the section on Constants, later in this chapter.

Operators

Pascal has a rich collection of data types and enables you to create your own data types. In contrast, Pascal has a relatively small collection of operations that you can carry out on variables. Other languages, such as C, have fewer data types but a greater number of operators.

An *operator* is something that can combine or change values to produce a new result, or that can produce a new value from an existing one. An operator generally takes information in one form and returns it in a different form.

For example, addition (+) takes information in the form of two values, and returns it in the form of a sum. Such an operator — which takes two values and returns one — is known as a *binary* operator.

The values that are associated with an operator are known as its *operands*. Operators and operands are combined in *expressions*. When you write an expression such as

```
7 + MyVal
```

the 7 is the *left operand* for the addition operator, and the variable **MyVal** is the *right operand*. Thus, a binary operator takes two operands and returns a single value.

There are also *unary* operators, which take one value and return another. For example, the **NOT** operator in Pascal takes an operand that is **TRUE** or **FALSE**, and returns the opposite value (that is, **FALSE** for **TRUE** and **TRUE** for **FALSE**). This operator takes a single operand (the original value) and returns a different value.

Certain operators are valid for all of the simple data types discussed in this chapter. However, in general, you cannot mix different types in the same expression. For example, you cannot compare an integer and a character directly in Pascal.

Pascal compilers are very good at detecting such *type conflicts*. With a few exceptions, type conflicts are not allowed in Pascal.

Assignment Operator

As you learned in Chapter 2, the assignment operator (**:=**) takes one operand — a variable or value — and assigns the value of this operand to a variable. The left element of an assignment statement must be a variable, since there must be storage in which to save the value. In general, the operand and the variable must be of the same type. The exceptions to this restriction will be noted as needed.

The following listing shows examples of assignment statements:

```
MyIntVariable := 37;

YourIntVariable := MyIntVariable;

MyString := 'hello';
```

```
MyValue := 7 * YourValue + 2 / TheirValue;
```

Notice that the operand for the assignment operator can be an extended expression. The entire expression is evaluated before the assignment operator is applied.

Comparison Operators

In programs, you often need to compare two values. For example, function **FindMax** in Chapter 2 compared **Val1** and **Val2**, to determine whether **Val1** was less than **Val2**. Similarly, you compared two strings in the **BubbleSort** procedure to determine whether one was "less" (came earlier in a sorted list) than another.

Such comparisons make sense for both numerical and nonnumerical information. The result of such a comparison is not a number. Rather, the result is a yes or a no — more commonly called true or false in computer science. (In fact, **TRUE** and **FALSE** are predefined values in Pascal.)

Pascal has six comparison, or relational, operators. Each of these operators takes two operands and returns a **TRUE** or **FALSE**. (Such values belong to a type known as **BOOLEAN**, which is discussed later in the chapter. Thus, comparison operators return **BOOLEAN** values.) While the comparison operators can handle both numeric and nonnumerical operands, the operands must be of the same type for any given comparison.

For instance, the first two comparisons in the following list are valid, but the last is not — because the two operands are of different types:

```
(* valid comparisons *)
'Hello' = 'Good Bye'
7 >= 23

7 < 'Hello'   (* invalid comparison: type conflict *)
```

Pascal's comparison operators are:

- Equality Operator (**=**). This operator returns **TRUE** if the two operands are equal to each other, **FALSE** otherwise. For example:

```
'hello' = 'hello'    (* TRUE *)
38 = 23              (* FALSE *)
```

- Inequality Operator (**<>**). This operator returns **TRUE** if the two operands are *not* equal to each other, **FALSE** otherwise. For example:

```
'hello' <> 'hello'    (* FALSE *)
38 <> 23              (* TRUE *)
```

- Greater than operator (**>**). This operator returns **TRUE** if the left operand is greater than the right one, **FALSE** otherwise. For example:

```
'zebra' > 'aardvark'    (* TRUE *)
'gnu' > 'gnu'           (* FALSE *)
38.65 > 199.357         (* FALSE *)
```

- Greater than or equal to operator (**> =**). This operator returns **TRUE** if the left operand is greater than *or equal to* the right one, **FALSE** otherwise. For example:

```
'zebra' >= 'aardvark'    (* TRUE *)
'gnu' >= 'gnu'           (* TRUE *)
38.65 >= 199.357         (* FALSE *)
```

- Less than operator (**<**). This operator returns **TRUE** if the left operand is less than the right one, **FALSE** otherwise. For example:

```
'zebra' < 'aardvark'    (* FALSE *)
'gnu' < 'gnu'           (* FALSE *)
38.65 < 199.357         (* TRUE *)
```

- Less than or equal to operator (**< =**). This operator returns **TRUE** if the left operand is less than *or equal to* the right one, **FALSE** otherwise. For example:

```
'zebra' <= 'aardvark'    (* FALSE *)
'gnu' <= 'gnu'           (* TRUE *)
38.65 <= 199.357         (* TRUE *)
```

TIP: The assignment and comparison operators are available for all of Pascal's simple data types. The comparison operators all produce a BOOLEAN result (a TRUE or FALSE), regardless of the operands being compared.

Operator Precedence

When you use more than one operator, *operator precedence* determines the order in which operators are applied.

For example, from algebra you might recall that multiplication takes precedence over addition. You can see this rule of precedence in the following expression, which evaluates to 26 rather than 42:

```
8 + 6 * 3
```

The result is 26 because the 6 * 3 (= 18) is computed first, and the resulting product is added to 8.

To make the expression equal 42, you need to add parentheses to override the rule of multiplicative precedence. The expression in the following listing evaluates to 42:

```
(8 + 6) * 3
```

Beyond such precedence rules, operators will generally be evaluated in a fixed direction. For example, most people evaluate the following expression from left to right, since all the operators have the same precedence.

```
1 + 2 + 6 + 23 + 85 + 189
    3 + 6 + 23 + 85 + 189
        9 + 23 + 85 + 189
            32 + 85 + 189
                117 + 189
                      306
```

Pascal's operators also have an order of precedence, which you'll learn as the operators are introduced.

The assignment operator has the lowest precedence of any Pascal operator. For this reason, you need not put parentheses around an expression on the operator's right hand side. In other words, the low precedence ensures that a value for the entire expression will be computed before that value is assigned to the variable on the left.

The comparison operators all have the same precedence, which is just *above* that for the assignment operator.

Constant Definitions

Often you'll need to use specific values in your programs. For example, suppose you're writing a physics program in which you need to describe the motion of falling bodies.

You will almost certainly need the value specifying the acceleration due to gravity (about 9.8 meters per second per second on earth). This value will be the same wherever you use it throughout the program. That is, the value will be *constant*.

Pascal lets you associate names with such values. You can then refer to the values by name. Such constant definitions belong in the declaration section of your program.

As with variable declarations, you can define multiple constants. The first line of constant definitions must begin with the reserved word **CONST**. Subsequent lines will begin with blank space or with identifiers.

Note that you use an equal sign, rather than an assignment operator, when defining a constant. This is deliberate. When you assign a value, you are carrying out an action — as implied by the term assignment *operator*. In contrast, when defining a constant, you are equating a name and a value, rather than storing a value in a particular memory location. Essentially, when processing your file, the compiler simply substitutes the specified value for every occurrence of the constant's name.

If the compiler simply replaces the name with the value, then why bother to define the constant in the first place? There are several reasons. First, your program will be much easier to understand if the reader sees something like **AccelerationConstant** instead of a value such as 9.8 in the source file.

Second, using a constant definition makes it much easier to change the value if necessary. For example, suppose your physics program used the value of the acceleration constant in 50 different places in the program. It would be very easy for you to simply type 9.8 in all of these places as you were writing your program.

However, suppose you want to adapt the program to do computations about bodies falling on the moon. There, the acceleration due to gravity is only about 1.6 meters per second per second. To adapt your program for moon motion, you would need to change 9.8 to 1.6 everywhere in your program.

A global search and replace may not work, since you may have used the same value (9.8) for a different purpose. In such a situation, the only course is to check each occurrence of 9.8 and decide whether to change it to 1.6. This method will be tedious and error prone. Chances are good that you'll overlook an occurrence, or change one too many values.

On the other hand, if you define a constant to represent the acceleration due to gravity, you can adapt your program by changing the value in the constant's definition. Thereafter, the compiler will make the changes for you. Doesn't it seem better to let the compiler do the work for you?

In standard Pascal, constant definitions must come before variable declarations in your program. In CodeWarrior Pascal, however, the relative order of constant definitions and variable declarations is not fixed.

Constant Expressions

CodeWarrior Pascal allows constant expressions — expressions that evaluate to a constant value. Such expressions can include literal values (numbers, characters, or strings), constant identifiers, operators. They may also include any of a limited number of functions: **ABS**, **CHR**, **ODD**, **ORD**, **ORD4**, **ROUND**, **SIZEOF**, **SQR**, **TRUNC**, or **UPCASE**.

The following are examples of constant expressions:

```
CONST Dozen = 12;
      Gross = Dozen * Dozen;
      BakersDozen = Dozen + 1;
      Yard = 3;
      SquareYard = SQR (Yard);
      CubicYard = Yard ** 3;
      SuitSet = [ club, diamond, heart, spade];
```

Variable Initialization

CodeWarrior Pascal lets you specify the initial values of program variables. However, you must do this as part of the Constant declaration section. Once you declare and initialize a variable, you needn't — in fact, you cannot — declare the variable again in the Variable declaration section.

To distinguish such variables from declarations of actual CONSTs, you must include type information about the variable. This information goes between the identifier and the equals sign. The type information is separated from the identifier by a colon (:), as in the following examples:

```
CONST ReallyAConst = 39;          (* Constant *)
      ReallyAVar : INTEGER = 17;  (* variable initialized to
17 *)
```

Predefined Procedures for Input and Output

Pascal provides procedures for reading and writing information. You've already used these procedures in the example programs in Chapter 2. In this section, we'll take a closer look at what these procedures are, and how to use them.

Output Procedures

Pascal provides the procedures **WRITE** and **WRITELN** for writing information to the screen or to a file. **WRITELN** displays the specified information and then moves the cursor to the start of the next line. (To move to the start of a new line, the procedure sends a combination of carriage return and line feed commands, often denoted by CRLF. Thus, **WRITELN** sends a CRLF after it has displayed all its arguments.) **WRITE** simply displays the specified information, but does not move the cursor to the next line (that is, does not send a CRLF). The following program shows how to call these two procedures.

Listing 3.5

```
(* Program to illustrate use of
   WRITE and WRITELN procedures.
*)

PROGRAM DisplayDemo;
VAR TestVal : INTEGER;
    Ch : CHAR;

BEGIN
  TestVal := 22222;(* Programmer should initialize variables *)
  WRITE ( 'Just a sentence, without CRLF.');
  WRITELN ( 'This continues on the same line.');
  WRITELN;
  WRITELN;
  WRITE ( 'There should be two blank lines above this one.');
  WRITELN;
  WRITE ( 'Eight = ', 8, '; nine = ', 9);
  WRITELN ( '; ten = ', 10);
  WRITELN ( 'TestVal = ', TestVal);
  WRITE ( 'Press ENTER to continue');
  READLN ( Ch);
END.
```

Run the program and compare its output with the following:

```
Just a sentence, without CRLF.This continues on the same line.
```

```
There should be two blank lines above this one.
Eight = 8; nine = 9; ten = 10
TestVal = 22222
Press ENTER to continue
```

Notice that **WRITE** and **WRITELN** can take a variable number of arguments. If you call **WRITELN** with no arguments, it simply sends its CRLF. (Although it is syntactically correct, you would never have reason to call **WRITE** without arguments, since it would then do nothing.)

Not only do these procedures take a variable number of arguments, the arguments can be of different types. For example, several of the calls have string arguments (words, phrases, or sentences within single quotes). Two of the last three procedure calls also have numerical values — namely, the numbers 8, 9, and 10 — as arguments. You can also pass variable names as arguments, as in the final call to **WRITELN**. In this case, the procedure writes the value currently stored at the location set aside for the variable.

If you are passing multiple arguments to **WRITE** or **WRITELN**, use commas to separate the arguments. You *need* not leave space between arguments, but you can leave as much as you wish. You may not put a space in the middle of an identifier. The following valid calls with multiple arguments both display the same results.

```
WRITE ( Arg1,Arg2,Arg3);
WRITELN ( Arg1,         Arg2,           Arg3);
```

Note that the space between arguments in the second statement does not put any space in the actual output.

You can continue an argument list over multiple lines; however, you must break between two arguments when doing this. In particular, you cannot break a string value over two lines. For example, the first of the following calls is valid, but not the second. The third call shows one way of fixing the incorrect call. It breaks the long string into two shorter ones, making each of these a separate argument.

```
(* a valid call to WRITELN *)
WRITELN ( Arg1, Arg2, 'A string argument',
          Arg4,
          'Another string argument', Arg6);

(* an invalid call to WRITELN *)
```

```
WRITELN ( Arg1, Arg2, 'A string argument
          broken over two lines',
          Arg4,
          'Another string argument', Arg6);

(* a possible fix for the preceding call to WRITELN *)
WRITELN ( Arg1, Arg2, 'A string argument',
          ' broken over two lines',
          Arg4,
          'Another string argument', Arg6);
```

To make your program easier to read, you may want to use space and line breaks between arguments in your procedure calls.

In Chapter 2, you also used **WRITELN** to write information to a file (**Log**) in the **Bubble** program. If the first argument to **WRITE** or **WRITELN** is a file variable, the information is written to the specified file; if no file variable is specified in the procedure call, the procedure writes to the standard output device, which is generally the screen.

In the sections on individual data types, you'll learn how to use **WRITE** and **WRITELN** to control the format of the information display.

Input Procedures

Pascal provides **READ** and **READLN** for reading input from the keyboard or from a file. As with the output routines, **READ** and **READLN** can take a variable number of arguments. For example, the following are all valid calls to these input routines:

```
READLN ( FileArg, MyInt, YourInt);

READ ( FileArg, MyInt, YourInt);

READLN ( Val1);
```

The first call reads two values from the file specified by **FileArg**, assuming that **FileArg** has been declared as a file variable. The first value read is assigned to the variable **MyInt** and the second is assigned to **YourInt**. After reading these two values, the procedure moves to the next line in the file.

The call to **READ** in the second example statement does the same thing as the first call, but it does not move to the next line in the file after reading the two values. This difference has consequences.

For example, suppose the information in your file is stored on lines as follows:

```
39  27   293
54
```

What will be the result for the following pair of calls? In particular, what will be the value of **TheirInt**?

```
READLN ( FileArg, MyInt, YourInt);
READLN ( FileArg, TheirInt).
```

After reading 39 into **MyInt** and 27 into **YourInt**, the first call to **READLN** moves to the next line in the file. Thus, 54 — the next integer value — is assigned to **TheirInt**. The value 293 is ignored in this example.

Suppose you had the following two calls instead:

```
READ ( FileArg, MyInt, YourInt);
READ ( FileArg, TheirInt).
```

In this case, 39 and 27 would be assigned as before. This time, however, 293 would be assigned to **TheirInt**, since the first call to **READ** did not move to the next line after reading its two values.

If your call to **READ** or **READLN** does not include a file variable as the first argument, the procedure assumes that the information is to come from the standard input, usually the keyboard. Thus, the following two calls each expect you to type in three values:

```
READ ( Arg1, Arg2, Arg3);
READLN ( Arg1, Arg2, Arg3);
```

If these are numeric values, you can enter some or all of the values on the same line — separated by spaces — or you can put the values on separate lines. You must press ENTER after putting the last value on each line.

If you simply call **READLN**, with no arguments, the program will stop when it gets to the procedure call, and will wait for you to press ENTER before continuing. This is a good way to have a program pause — for example, while you're examining output.

If you tell **READ** or **READLN** to read a particular type of value, and then type a different kind of value, you may get a run-time error and you may crash your program. For example, if you tell the procedure to expect a number, and it finds a string of characters instead, you'll get a run-time error.

Real Numbers

Sometimes you'll be able to do your work using just whole numbers. At other times, you'll need to work with numbers (such as 1.5 or 39.33) that have fractional components. Such "fractional" numbers are called *real numbers*.

Real numbers use a period (.) to separate the whole number from the fractional part. Ordinarily, this period is called a decimal point, because we generally represent numbers using 10 as the base. However, it is possible to use bases other than 10 to represent numbers. For example, most compilers represent values in binary form — using 2 as the numerical base.

This discussion uses decimal values, but the same concepts also apply when you use other bases.

Pascal's *REAL* ***Type***

Pascal has a predefined type, **REAL**, for representing real numbers. The CodeWarrior Pascal compiler allocates four bytes for each **REAL** variable. Within this storage, the range of possible **REAL** values you can represent is:

1.401e-45 through 3.403e38 for positive numbers, and

-1.401e-45 through -3.403e38 for negative numbers.

The following program shows how to declare and use variables of type **REAL**. Try several different values when running the program. Enter some very small values and also some extremely large values. You might try entering a value with a large number of decimal digits.

Listing 3.6

```
(* Program to illustrate declaration and use of REALs *)
PROGRAM RealTest;

VAR RValue : REAL;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Get a REAL value from user. *)
  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN   (* Main Program *)
  GetReal ( 'Value (< 0 to stop) ', RValue);
  WHILE RValue >= 0 DO
  BEGIN
    WRITELN ( RValue);          (* use default output format *)
    WRITELN ( RValue : 10 : 5); (* use specified output format *)
    GetReal ( 'Value (< 0 to stop) ', RValue);
  END; (* WHILE RValue >= 0 *)
  AwaitUser;
END.
```

This program gets a **REAL** value from the user, and then displays the value twice. The output format is discussed later in the chapter.

When it sees the declaration for the variable **RValue**, the compiler sets aside four bytes of storage, because **RValue** is declared to be a **REAL**.

Procedure **GetReal** is modelled on **GetString**, which was used in Chapter 2.

Syntax for REAL Values

REAL values must have a particular format in Pascal. A **REAL** value may begin with an optional sign (+ or -). For example, +35.35, -3.53, and 353.5 are all valid **REAL**s.

A **REAL** value *must* contain at least one digit, and *may* contain one decimal point. If the number contains a decimal point, there must be *at least* one digit before (to the left of) the decimal point. (This digit can be 0 if necessary.) For example, the first six values in the following listing are valid, but the remaining ones are invalid:

```
(* valid real numbers *)
35.47
+5.5
-6
399999
0.731
-3.

(* invalid real numbers *)
+.3          (* no digit before decimal point *)
3.+5         (* sign is not at start of value *)
3.5.7        (* only one decimal point allowed per number *)
```

You may use scientific notation to specify real numbers. In *scientific notation*, numbers are expressed as a value and an exponent. The *exponent* specifies the power of 10 by which you need to multiply the value in order to get the number. For example, the following listing represents 15 in several different ways using scientific notation, and also illustrates the syntax for scientific notation in Pascal:

```
0.15E2
1.5E1
15e0
150e-1
1500E-2
```

The *e* (or *E*, since Pascal is not case sensitive) marks the beginning of the exponent. The value following this represents the power of 10 by which you need to multiply the number. Each positive power of 10 means another multiplication by 10. Thus, **0.15E2** means 0.15 * 10^2, or 0.15 * 10 * 10, or 0.15 * 100.

Similarly, each *negative* power of 10 means to *divide* by 10. Thus, **1500E-2** means 1500 / 10 / 10 or 1500 / 100.

If you use scientific notation, the *e* may be followed by a sign (+ or -), and *must* be followed by a whole number.

The format for a valid REAL value is as follows:

Optional + or - sign

One or more digits (which may be 0)

Optional decimal point followed by one or more digits

Optional *E* or *e*, followed by optional + or - sign, followed by a whole number

---

Displaying REAL *Values*

When displaying **REAL** values, you can control the output format or you can let the system use its default display format. The first call to **WRITELN** in the **RealTest** program uses the default output format for **REAL** values. By default, CodeWarrior Pascal writes real numbers in scientific notation.

For example, for an input of 3579.987654, the earlier program **RealTest** produces the following output:

```
 3.580E+03
3579.98755
```

The first line contains the scientific notation. Recall that the number following the *E* is the power of 10 by which the value needs to be multiplied. An exponent value of 3 means to multiply the number by 10^3, or 1000.

The second call to **WRITELN** in **RealTest** also displays the value of **RValue**, but uses a format you specify. In the call to **WRITELN**, the

```
: 10 : 5
```

after the variable name specify the field width and the precision for the output.

- The *field width* specifies the number of columns to use when displaying the number. The field width for the output is set to 10 columns.
- The *precision* specifies the number of digits to include after the decimal point. The precision is set to 5 digits after the decimal point.

The decimal point and sign are included in the field width. Thus, when you specify a field width of 10 columns, the decimal point and sign may use two of these. In that case, you would get only eight digits.

If the value to be written is longer than the field width you've specified, **WRITELN** ignores your format instructions, and uses as many columns as needed. If fewer columns are needed than you've specified, the output has blanks at the left; the rightmost digit is in the last column specified by the field width.

REAL ***Values as Approximations***

You may recall from algebra that there are an infinite number of different real values. In fact, you can find as many values as you wish between any two real values. However, there are only a finite number of bit combinations possible in the storage space allocated for a **REAL** variable. This means that not all real numbers can be represented by a variable of type **REAL**. Such values are actually approximate values.

The approximations can be in error for any of several reasons. There may not be enough bits to represent the required number of digits. For example, a **REAL** in CodeWarrior Pascal can accurately represent values having no more than about 8 digits.

Errors may be introduced because of rounding or truncation that take place during computations. For example, suppose you want to add the following

0.12345678901234567890 and 1.12345678901234567890

Since these values exceed the limits of precision (about 8 digits) for type **REAL**, your system will actually perform a computation involving fewer digits. The result will reflect these truncated computations.

This means that some computations involving **REAL** variables and values will be only approximately correct, particularly computations that involve an extreme range of values, values requiring extreme precision, or values requiring lots of computations.

Fortunately, such errors will be very small in magnitude, and generally will appear only in the extreme decimal places — at values smaller than the precision you'll need for most computations.

When comparing two **REAL** values, it is generally better to check whether the two differ by less than some small amount, rather than whether the two values are exactly equal.

WARNING! Be careful when using REAL variables, especially when looking for exact values. The representations for two apparently identical REAL values may actually be different because of the way the two values were computed.

Operators for REAL *Values*

In addition to specifying the amount of storage, a variable declaration determines the operations you can carry out on the variable. As you'll find, certain operators make sense only with certain variables. For example, you can add or multiply two numbers, but not two characters.

Assignment and Comparison Operators

You can use the assignment operator and the comparison operators with **REAL** variables and values. The following program illustrates these operators:

Listing 3.7

```
(* Program to illustrate use of
   assignment and comparison operators with REALs
*)

PROGRAM RealOps1;
```

```
CONST CutOff = 1000.0;
      TwiceCutOff = 2000.0;
VAR Val1, Val2 : REAL;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN    (* Main Program *)
  GetReal ( 'Value 1:', Val1);
  IF Val1 > CutOff THEN
    Val2 := Val1
  ELSE
    Val2 := CutOff;
  WRITELN ( 'Val1 = ', Val1 : 10 : 3,
            '     Val2 = ', Val2 : 10 : 3);
  IF Val1 <= TwiceCutOff THEN
    Val2 := TwiceCutoff;
  WRITELN ( 'Val1 = ', Val1 : 10 : 3,
            '     Val2 = ', Val2 : 10 : 3);
  AwaitUser;
END.
```

The main program uses two comparison operators (both in **IF** clauses) and three assignment operators (all of which change the value of **Val2**). For inputs of 123.45, 1234.5, and 12345.0, the program produced the outputs shown in the following listing.

```
(* for 123.45 *)
Val1 =    123.450     Val2 =   1000.000
Val1 =    123.450     Val2 =   2000.000

(* for 1234.5 *)
Val1 =   1234.500     Val2 =   1234.500
Val1 =   1234.500     Val2 =   2000.000

(* for 12345.0 *)
Val1 =  12345.000     Val2 =  12345.000
Val1 =  12345.000     Val2 =  12345.000
Press ENTER to continue
```

As mentioned, it can be dangerous to check whether two **REAL** values are exactly equal. The following program illustrates this.

Listing 3.8

```
(* Program to illustrate dangers of comparing REALs
   for strict equality.
*)

PROGRAM PrecisionDemo;

CONST Increment = 0.003;
VAR CompVal : REAL;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
```

```
  CompVal := 0.9;
  WRITELN ( 'Before loop, CompVal = ', CompVal : 10 : 7);

  AwaitUser;
  (* Loop will keep executing, because the bit patterns
     for CompVal and 1.2 almost certainly won't be identical.
     Press CTRL-C to end program.
  *)
  WHILE ( CompVal <> 1.2) DO
  BEGIN
    WRITELN ( 'CompVal = ', CompVal : 10 : 7);
    CompVal := CompVal + Increment;
    IF ( CompVal > 1.2) THEN
      READLN;
  END;
  WRITELN ( 'CompVal = ', CompVal : 10 : 7);
  AwaitUser;
END.
```

This program executes a loop until you break out of the program, which you can do by pressing CTRL-C.

The program *apparently* reaches 1.2, the value that should stop the **WHILE** loop. The output indicates that **CompVal** is equal to 1.2 — at least to five decimal places. However, the fact that the loop continues executing indicates that the two values are not exactly identical.

When checking for the equality (or inequality, as in this case) of two **REAL**s, CodeWarrior Pascal compares their actual bit patterns. The value of **CompVal** is attained through repeated computations, which can introduce errors. Its bit pattern will almost certainly differ from the pattern for 1.2.

This is why it's extremely unlikely that two **REAL**s will be exactly equal — especially if either or both values have been obtained through computations.

Arithmetic Operators

Not surprisingly, you can carry out the usual operations on **REAL** values: You can add, subtract, multiply, and divide them. CodeWarrior Pascal also supports an exponentiation operator that enables you to raise one number to a power specified by a second value.

The addition (+) and subtraction (-) operators enable you to combine two **REAL** values to produce a third value of the same type. The following program lets you add or subtract two values. Run the program, and try several pairs of values.

Listing 3.9

```
(* Program to illustrate use of
   addition and subtraction operators for REALs
*)

PROGRAM RealOps2;

CONST EmptyString = '';    (* a blank string *)
      PlusOp = '+';
VAR Val1, Val2 : REAL;
    Operation : STRING;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)
```

```
BEGIN   (* Main Program *)
  (* Find out what operator user wants *)
  GetString ( '+ or - (ENTER to stop)', Operation);
  WRITELN ( 'Operation = ', Operation);
  (* While user wants to continue ... *)
  WHILE Operation <> EmptyString DO
  BEGIN
    GetReal ( 'Value 1: ', Val1);
    GetReal ( 'Value 2: ', Val2);
    IF Operation = PlusOp THEN
      WRITELN ( Val1 + Val2 : 10 : 3)
    ELSE
      WRITELN ( Val1 - Val2 : 10 : 3);
    GetString ( '+ or - (ENTER to stop)', Operation);
    WRITELN ( 'Operation = ', Operation);
  END; (* WHILE Operation <> EmptyString *)
  AwaitUser;
END.
```

Notice the arguments in the calls to **WRITELN** in the main program. Despite the length of the material within parentheses, these calls have just one argument. The expressions,

```
Val1 + Val2

Val1 - Val2
```

each evaluate to a **REAL**, which is substituted for the expression before the writing occurs. The remaining material within parentheses specifies the format.

Note the constant, **EmptyString**. This is defined as a string with no characters — as when you press ENTER without typing anything else first.

The multiplication (*) and division (/) operators also work as they do in algebra. Recall that division by 0 is not defined in algebra. In standard Pascal, division by zero produces a run-time error, and your program crashes; in CodeWarrior Pascal, such a division produces an answer of **INF** (for infinity), and your program continues.

The following program lets you multiply or divide two values:

Listing 3.10

```
(* Program to illustrate use of
   multiplication and division operators for REALs
*)

PROGRAM RealOps3;

CONST EmptyString = '';
      TimesOp = '*';
VAR Val1, Val2 : REAL;
    Operation : STRING;

  (* Wait for user to press ENTER *)
   PROCEDURE AwaitUser;
   VAR Ch : CHAR;
   BEGIN
     WRITE ( 'Press ENTER to continue.');
     READLN ( Ch);
   END; (* AwaitUser *)

   PROCEDURE GetString ( Message : STRING;
                         VAR Value : STRING);
   BEGIN
     WRITE ( Message, ' ');
     READLN ( Value);
   END; (* GetString *)

   PROCEDURE GetReal ( Message : STRING;
                       VAR Value : REAL);
   BEGIN
     WRITE ( Message, ' ');
     READLN ( Value);
   END; (* GetReal *)
```

```
BEGIN   (* Main Program *)
  (* Find out what operator user wants *)
  GetString ( '* or / (ENTER to stop)', Operation);
  WRITELN ( 'Operation = ', Operation);
  (* While user wants to continue ... *)
  WHILE Operation <> EmptyString DO
  BEGIN
    GetReal ( 'Value 1: ', Val1);
    GetReal ( 'Value 2: ', Val2);
    IF Operation = TimesOp THEN
      WRITELN ( Val1 * Val2 : 10 : 3)
    ELSE
      WRITELN ( Val1 / Val2 : 10 : 3);
    GetString ( '* or / (ENTER to stop)', Operation);
    WRITELN ( 'Operation = ', Operation);
  END; (* WHILE Operation <> EmptyString *)
  AwaitUser;
END.
```

This program is actually identical to the preceding one, with the following exceptions:

The constant **PlusOp** has been replaced by **TimesOp**.

- The operators in the message for both calls to **GetString** have been changed (from + to * and from - to /).
- The operators in the two calls to **WRITELN** in the main program have been changed.

CodeWarrior Pascal also includes an exponentiation operator (**). Using this argument, you can raise the left operand (the *base* to the power specified by the right operand (the *exponent*). For example, consider the following exponentiation expressions.

```
6 ** 3      (* evaluates to the cube of 6:
               that is, 6 * 6 * 6, or 256 *)
49 ** 0.5   (* evaluates to the square root of 49, or 7 *)
10 ** -2    (* evaluates to 1/100 *)
```

The following program lets you explore the exponentiation operator.

Listing 3.11

```
(* Program to illustrate use of
   exponentiation operator for REALs
*)

PROGRAM RealOps4;

CONST EmptyString = '';
      ExpOp = '^';
VAR Val1, Val2 : REAL;
    Operation : STRING;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)
```

```
BEGIN   (* Main Program *)
  GetString ( '^ to exponentiate (ENTER to end loop)',
Operation);
   (* While user wants to continue ... *)
  WHILE Operation <> EmptyString DO
  BEGIN
    GetReal ( 'Value 1: ', Val1);
    GetReal ( 'Value 2: ', Val2);
    IF Operation = ExpOp THEN
      WRITELN ( Val1 :10 : 3, ' ** ',  Val2 : 10 : 3, ' = ',
                Val1 ** Val2 : 10 : 3)
    ELSE
      WRITELN ( 'No operator selected.');
    GetString ( '^ to exponentiate (ENTER to end loop)',
Operation);
  END; (* WHILE Operation <> EmptyString *)
  AwaitUser;
END.
```

Try various kinds of values. In particular try using the following kinds of values:

- Positive whole numbers and numbers with fractional components for the base, and positive whole numbers for the exponent.
- 0 for the exponent.
- 0 for the base.
- A negative whole number for the exponent.
- Negative base and even or odd whole numbers for the exponent.
- Negative base and numbers with fractional components for the exponent.

The last example in the list will produce **NAN** (for "not a number") as output because such values are not allowed. For example, -2 ** 0.5 would be equivalent to computing the square root of negative 2. As you may recall from algebra, this value does not exist among the real numbers.

Thus, the exponentiation operator allows only whole number exponents if the base value is less than zero.

Precedence of Operators for REALs

The arithmetic operators actually fall into two levels of precedence. Multiplication, exponentiation, and division — all known as *multiplicative operators* (division is merely the inverse of multiplication) — have the same precedence. This precedence is higher than that for the *additive operators* (addition and subtraction).

The precedence of all arithmetic operators is higher than that of the comparison operators and, therefore, of the assignment operator. Later sections extend this order of precedence to include other operators.

Miscellaneous Issues Concerning REALs

In standard Pascal, if you accidentally type a nonnumerical value when the program is expecting a **REAL**, the program will crash with a run-time error. CodeWarrior Pascal will convert the input to a zero value, and the program will continue with that as the value for the variable in question.

Similarly, if a computation tries to perform a mathematically illegal operations (such as dividing by zero or raising a negative number to a fractional power), a CodeWarrior Pascal program returns **NAN** (for not a number), along with an error code in parentheses. Programs compiled using standard Pascal will crash with a run-time error in such cases.

Although Pascal's number representation lets you represent a large range of **REAL** values, you can still exceed these limits. This is known as *overflow*. In case of overflow involving **REAL** values, a program written in standard Pascal will crash with a run-time error. In contrast, a CodeWarrior Pascal program will return **INF** or **-INF** (for infinity or negative infinity) in cases where the magnitude of the computed value is too large to be represented in CodeWarrior Pascal's number system.

Variants and Extensions of the REAL Type

In addition to type **REAL**, whose variables are allocated four bytes each, CodeWarrior Pascal supports type **LONGREAL**. Variables of this type are allocated eight bytes each by default, but may be allocated 10 or even 12 bytes in some environments.

Type **REAL** may also be known as type **SINGLE** (for single precision). Similarly, type **LONGREAL** may be known as type **DOUBLE** (for double precision).

Whole Number Types

CodeWarrior Pascal provides data types for storing whole numbers. For the most part, these whole number types differ in the amount of storage allocated for the type, and in whether the type allows negative numbers.

The INTEGER *Type*

The **INTEGER** type is defined for all versions of Pascal. This type is used to represent whole numbers from -32768 to 32767. The compiler allocates two bytes, or 16 bits, for values of type **INTEGER**. Within this representation, you can store values ranging from -2^{15} through $2^{15} - 1$, inclusive. (The leftmost, sixteenth, bit is used to represent the value's sign. If this bit is 1, the value is negative.)

Pascal includes a predefined constant, **MAXINT**, whose value is the maximum integer value for the implementation. In CodeWarrior Pascal, this value is 32767.

The following program shows how to declare, initialize, and display **INTEGER** variables:

Listing 3.12

```
(* Program to illustrate use of integer values *)

PROGRAM IntExample;

VAR Int1, Int2 : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Get an integer value from the user. *)
  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
```

```
    WRITE ( Message, '? ');    (* prompt user *)
    READLN ( Value);           (* read response *)
  END; (* GetInteger *)

BEGIN
  GetInteger ( 'Value 1', Int1);
  GetInteger ( 'Value 2', Int2);
  WRITELN ( 'The sum of ', Int1, ' and ', Int2,
            ' is ', Int1 + Int2);
  WRITELN ( 'MAXINT = ', MAXINT);
  AwaitUser;
END.
```

As usual, the variable declarations are made in the declaration section of the program. In this case, two integer variables are declared and the compiler sets aside four bytes of storage for these two variables.

Like **GetReal**, procedure **GetInteger** is modeled on **GetString** in Chapter 2. You'll build an improved version of this procedure in Chapter 6.

When specifying integers, you cannot use commas between digits; you may use only digits and an optional sign at the start of the number. For example, -32500 and +28400 are valid, but -32,500 and +28,400 are not.

Controlling Output Format

CodeWarrior Pascal differs from standard Pascal in its handling of whole number output. By default, standard Pascal uses only as many columns as needed to write your whole number value. This is convenient, but can also have unexpected consequences. For example, in standard Pascal, the output from the following program would run the three values together.

Listing 3.13

```
(* Program to illustrate what happens when you don't
   leave any space between values when writing them.
 ********************************************************
```

```
   NOTE: CodeWarrior Pascal uses a default of 8 columns
         if no minimum field width is specified.
         As a result, the output would not be run together
         in a default CodeWarrior compilation of this program.
         By using : 0 to format the output, you get the
         output Standard Pascal would give.
*)

PROGRAM NoSpace;

VAR Val1, Val2, Val3 : INTEGER;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN
  GetInteger ( 'Value 1', Val1);
  GetInteger ( 'Value 2', Val2);
  GetInteger ( 'Value 3', Val3);
  WRITELN ( Val1 : 0, Val2 : 0, Val3 : 0);
  AwaitUser;
END.
```

The following listing shows the output *from a standard Pascal* program and from a CodeWarrior program that overrides default formats — for inputs of 33, 66, and 99:

```
336699
```

To avoid this error in standard Pascal, you can explicitly output blanks between the values. The following version of the call to **WRITELN** will put space between the values:

```
WRITELN ( Val1, '    ', Val2, '    ', Val3);
```

You can also specify a different output format, if you wish. To change the format, you need to specify the number of columns the program should use when writing your value. For example, the following statement says to write the value of **ValToDisplay**, and specifies a field width of 15 columns:

```
WRITELN ( ValToDisplay : 15);
```

If the value being written is shorter than the specified field width, the left part of the output is filled with blanks. Such an output format — in which the last digit of the value is in the last column specified by the field width — is said to be *right justified*, or *flush right*.

If the value being written is longer than the field width, the field width is ignored and the entire value is written. Effectively, standard Pascal uses a default field width of 0 when displaying whole number values. Since any actual output will be longer than this, the preset field width will be overridden, and the output will use up just the required number of columns.

Output Formats in CodeWarrior Pascal

In contrast to standard Pascal, CodeWarrior Pascal uses a default field width of 8 when displaying whole numbers. As a result, the output from the preceding program will look different if you compile it under CodeWarrior Pascal.

As with standard Pascal, however, you can override the default field width. For example, if you specify a 0 field width, then CodeWarrior Pascal's output will look like standard Pascal's. To see this, edit the preceding program by changing the **WRITELN** in the main program to the following version:

```
WRITELN ( Val1 : 0, Val2 : 0, Val3 : 0);
```

Operations on INTEGERs

In addition to the assignment and comparison operators (which are available for all types), you can use the following operators with **INTEGER** operands.

Arithmetic Operators

You can add, subtract, and multiply **INTEGER** values. In CodeWarrior Pascal, you can also use the exponentiation operator to raise integers to an integral power. These operators work just as they do for **REAL** operands.

To add two integers, just use the addition operator (+). The examples in the following listing use this operator:

```
MySum := IntVal1 + IntVal2;

WRITELN ( MySum + YourSum);

Total := MySum + YourSum + TheirSum;
```

Similarly, the subtraction operator (-) lets you subtract one value from another, as you can see in the following listing:

```
MyDifference := IntVal1 - IntVal2;

WRITELN ( MyDifference - YourDifference);

Total := MyDifference - YourDifference - TheirDifference;
```

The multiplication operator (*) produces a result that is the product of its two operands.

```
MyProduct := IntVal1 * IntVal2;

WRITELN ( MyProduct * YourProduct);

(* The following could be dangerous because the
   products may be too big for the integer range.
   See text for more on integer overflow.
*)
Total := MyProduct * YourProduct * TheirProduct;
```

Finally, the exponentiation operator (**) for whole numbers raises one whole number to a power specified by another. For example,

```
2 ** 7 = 2 * 2 * 2 * 2 * 2 * 2 * 2 = 128

4 ** 3 = 4 * 4 * 4 = 64
```

Because of the limitations on the range of values an **INTEGER** variable can take on, you always need to make sure you're not going beyond the maximum **INTEGER** values (-32768 and 32767 in current implementations). It's not hard to do this: for example, 200 * 200 is 40,000. This value cannot be represented correctly in an **INTEGER** variable. Overflow becomes an issue even more quickly with the exponentiation operator.

Either you need to use smaller numbers or you need to use the **LONGINT** data type, which is supported by CodeWarrior Pascal, but which is not part of standard Pascal.

What about division? What's 17 divided by 3 when you're working with whole numbers? As you may recall, in such a case, the result is generally expressed as a whole number value and a remainder. For example, when you divide 17 by 3 the result is 5 with a remainder of 2. 3 goes into 17 five times, since 3 * 5 = 15; the 2 left over after subtracting 15 from 17 is the remainder.

Pascal has two operators to produce the preceding results for integers. The division operator for **INTEGER** operands is **DIV**. This operator takes two integer operands — for instance, **Op1** and **Op2** — and returns the number of times **Op2** goes completely into **Op1**. The **DIV** operator ignores any remainder.

The following program demonstrates the **DIV** operator:

Listing 3.14

```
(* Program to exercise INTEGER DIV operator *)

PROGRAM TestIntOps;

CONST IntErrorValue = -32768;
      Op = 'DIV';
VAR   IntResult, Val1, Val2 : INTEGER;
      Count : INTEGER;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

   (* Return result of applying DIV operator to arguments *)
  FUNCTION IntDiv ( Dividend, Divisor : INTEGER) : INTEGER;
  BEGIN
    IF Divisor = 0 THEN
      IntDiv := IntErrorValue
    ELSE
      IntDiv := Dividend DIV Divisor;
  END;  (* IntDiv *)

BEGIN
  GetInt ( 'Value 1', Val1);
  GetInt ( 'Value 2', Val2);
  FOR Count := 1 TO 3 DO
  BEGIN
    IntResult := IntDiv ( Val1, Val2);
    WRITELN ( Val1, ' ', Op, ' ', Val2, ' = ', IntResult);
    Val1 := IntResult;
  END; (* For count := 1 to 3 *)
  AwaitUser;
END.
```

For inputs of 178 and 7 as **Val1**, and **Val2**, respectively, the program produces the following output in the CodeWarrior environment:

```
178 DIV         7 =        25
 25 DIV         7 =         3
  3 DIV         7 =         0
```

As you saw, integer division produces two results: a whole number quotient and a remainder. The **MOD** operator return the remainder.

Given two **INTEGER** arguments, the **MOD** operator returns the whole number remainder when the first argument is divided by the second (using integer division). For example, when **LeftOperand** is 78 in the following listing, the expression

```
LeftOperand MOD 23
```

evaluates to 9, since 78 - 69 = 9. Notice that the result from applying the **MOD** operator will range between 0 and one less than the second (right) operand. Thus, the result of the expression in the preceding listing — that is, one computing MOD 23 — will always be a value between 0 and 22.

The following program illustrates the **MOD** operator:

Listing 3.15

```
(* Program to illustrate use of MOD operator *)

PROGRAM ExploreMod;
USES Windows;

CONST MaxTrials = 500;
VAR Result, Count, HeadCount, RowCount, RandSeed : INTEGER;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN
  (* Get and set seed value for random number generator *)
  GetInt ( 'Integer seed for random number generator', RandSeed);
  SRAND ( RandSeed);
  HeadCount := 0;
  RowCount := 0;
  FOR Count := 1 TO MaxTrials DO
  BEGIN
    (* the following call returns a pseudorandom integer
       between 0 and MAXINT
    *)
    (* Result := RANDOM ( MAXINT); *)
    Result := RAND;
    (* If result is even, count it as a head,
       and write a period.
    *)
    IF result MOD 2 = 0 THEN
    BEGIN
      WRITE ( '.');
      RowCount := RowCount + 1;
    END
    ELSE   (* if not even, write a space *)
      WRITE ( ' ');
    (* after 50 trials, move to a new line *)
    IF Count MOD 50 = 0 THEN
    BEGIN
      WRITELN ( RowCount : 3);
      HeadCount := RowCount + HeadCount;
      RowCount := 0;
    END;  (* IF count MOD 50 = 0 *)
  END;
  WRITELN ( HeadCount, ' heads');
  AwaitUser;
END.
```

This program uses random number functions that require the **Windows** unit. To include this unit in the source file, you just need to include the **USES Windows;** line after the **PROGRAM** line. When creating your project, you need to include several files in addition to the usual library files: **WINDOWS.P**, CONSTANTS.P, **FUNCTIONS.P**, MACROS.P, **TYPES.P**, and **VARIABLES.P**.

The call to the predefined procedure **RAND** returns a pseudorandom integer in the range 0 through **MAXINT**. The call to **SRAND** sets a seed for **RAND** to use when generating its values. By setting a different seed, you can get different random number sequences; conversely, by setting the same seed, you can replicate sequences of generated values. Note that this program uses the **Windows** unit, which is needed for the random number routines.

If the value returned is even, the program writes a period and increments a counter. If the value is not even, the program writes a space. Every 50 trials, the program moves to a new line. This is determined by testing whether **Count** leaves a remainder of 0 when divided by 50.

For a sample run, the program produced the following output:

```
... .  . .. ..  . .. .......  ... ..  ..... ....   33
..  .  . ... .. ...   .  . .      .   . . .  . ..  22
.   .. . ..  ..  . . .    ..  . .. ..      .... .  23
  ......     ...   .  .. .. . .    ... .  .. ... . 26
.. . .. .     .  ... ..   . ... . . ..... ..   ... 28
..  .  .. . . .. .   .. .. . .  .  ..  . . .  . .. 25
.  .....  .     . .   ...   ....  ..  . . . . . .. 25
..  ..     . ....    .... .  .. . .... .    . .. . 26
 .. .     . .. ..  . . .  ..  .... .   .. ....     24
     . .  ..  .  .... .. ..... ....... ... . ..... 32
264 heads
```

Precedence for the Arithmetic Operators

Since the addition, subtraction, exponentiation, and multiplication operators are the same ones as for **REAL**s, their precedence is the same as for that type.

Both the **DIV** and the **MOD** operators are involved in division which, as you saw earlier, is a multiplicative operator. Not surprisingly, the precedence for the **DIV** and **MOD** operators is the same as for the multiplication and (**REAL**) division operators.

Hexadecimal Values

In everyday work, you deal with numbers in decimal form, using 10 as the numeric base. In base 10, you work with 10 different digits: 0 through 9. In decimal values, each column represents a power of 10. In particular, each column represents values that are 10 times the values to the right. Thus, the decimal value 4326 is composed of 6 ones, 2 tens, 3 hundreds, and 4 thousands.

In computer science, numbers are often represented in bases other than 10. The most common alternate representations are binary (base 2), octal (base 8), and hexadecimal (base 16).

In base 16, each column represents a different power of *16,* rather than 10. Thus, the hexadecimal number 4326 represents 6 ones, 2 sixteens, 3 two-hundred-fifty-sixes (16 * 16), and 4 four-thousand-ninety-sixes (256 * 16). Converted to decimal form, this value is 6 + 32 + 768 + 16,384, which equals 17,190.

In base 10, you have 10 different digits; you need 16 digits for base 16. To represent such values, use the digits 0 through 9 as usual; then use the letters *a* through *f* (or *A* through *F*) to represent the hexadecimal digits 10, 11, 12, 13, 14, and 15, respectively. Thus, the hexadecimal value *CAFE* is equivalent to *12 * 4096 + 10 * 256 + 15 * 16 + 14*, or 51,966.

You can use hexadecimal values in your CodeWarrior Pascal programs. To specify that a value is hexadecimal, rather than decimal, put a dollar sign ($) before the value. For example, $39 is equivalent to 57 in base 10 (3 * 16 + 9). The following program shows how to assign hexadecimal values.

Listing 3.16

```
(* Program to illustrate use of hexadecimal constants. *)

PROGRAM HexDemo;

VAR HexInt : INTEGER;

  (* Wait for user to press ENTER *)
   PROCEDURE AwaitUser;
   VAR Ch : CHAR;
   BEGIN
     WRITE ( 'Press ENTER to continue.');
     READLN ( Ch);
   END; (* AwaitUser *)

BEGIN
  HexInt := $387d;
  WRITELN ( 'In decimal form, HexInt = ', HexInt);
  AwaitUser;
END.
```

Miscellaneous Issues Concerning INTEGERs

Whole numbers are a special case of real numbers — they're simply real numbers without fractional parts. Therefore, you can assign an **INTEGER** value to a **REAL** variable without complaint from the compiler. You cannot do the reverse, however. That is, you can't assign a **REAL** to an **INTEGER** variable. The compiler will report a type conflict error.

What happens if you try to represent a value outside the range of possible values? For example, what happens if you add 1 to 32767 (that is, **MAXINT**)? Recall that overflow with

REAL values results in a run-time error in standard Pascal and in a value of **INF** in CodeWarrior Pascal.

INTEGER overflow does not necessarily produce a run-time error. However, such overflow will produce an incorrect result.

The following program illustrates **INTEGER** overflow: Try adding and subtracting various values. What happens when you add values that are several times too large (for example adding 400000 to a value)?

Listing 3.17

```
(* Program for exploring integer overflow *)

PROGRAM Test;

CONST Smallest = -32768;
      Largest = MAXINT;

VAR FromSmallest, ToLargest : INTEGER;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN
  GetInteger ( 'Subtract what from smallest? (0 to stop)',
               FromSmallest);
  (* while user wants to explore,
     subtract from Smallest and add to Largest
  *)
  WHILE FromSmallest <> 0  DO
  BEGIN
    GetInteger ( 'Add what to largest?',
                 ToLargest);
    WRITE ( Smallest:7, ' - ', FromSmallest:7, ' = ');
    WRITELN ( Smallest - FromSmallest: 7);
    WRITE ( Largest:7, ' + ', ToLargest:7, ' = ');
    WRITELN ( Largest + ToLargest: 7);
    GetInteger ( 'Subtract what from smallest? (0 to stop)',
                 FromSmallest);
  END; (* WHILE FromSmallest <> 0 *)

  AwaitUser;
END.
```

Also, build the program with the "Activate Overflow Checking" Pascal compiler setting turned on and turned off. When this is off, the compiler will not include any checks for overflow violations. Unless you turn on overflow checking, you will not get any error messages when integer overflow occurs at run-time — as part of a computation or through an assignment. Therefore, you need to be very careful when multiplying integers or when working with values near the range limits.

For example, with overflow checking turned off, the program produces the following for inputs of **5** for both **FromSmallest** and **ToLargest**:

```
Subtract what from smallest? (0 to stop) 5
Add what to largest? 5
 -32768 -       5 =   32773
  32767 +       5 =  -32764
Press ENTER to continue
```

By default, the CodeWarrior Pascal compiler does not check for overflow; you can turn the compiler's overflow checking on, however — in the Pascal Compiler part of the Project settings. If you do check for overflow, you'll get a run-time error when using the values in the previous example.

The Pascal Compiler settings also include a range checking option. If you turn this on, the compiler will catch assignments that exceed the range of valid values for INTEGERs. For example, trying to assign a value of 35000 (which exceeds the maximum value of 32767) will result in a compiler error.

If large values are likely to occur in your program, you should consider working with **REAL** variables, or with one of the longer integer variants described in the next section.

Variants and Extensions of the INTEGER Type

In addition to type **INTEGER**, which is the only whole number type defined in standard Pascal, CodeWarrior Pascal provides several other simple types for representing whole numbers. These types differ from **INTEGER**s in the range of values that can be represented.

The same operators are available for these types as for **INTEGER** variables. Any differences in the way overflow is handled are discussed in connection with the individual types.

Type **LONGINT**

CodeWarrior Pascal provides a **LONGINT** type for representing a larger range of values than that afforded by the **INTEGER** type. The compiler allocates four bytes (32 bits) for **LONGINT**s. This gives a range of possible values between -2^{31} and $2^{31} - 1$, or between -2,147,483,648 and 2,147,483,647, inclusive. (The commas in the numbers are for clarity. You can't use commas when writing **LONGINT**s in your programs.)

The predefined constant, **MAXLONGINT**, is set equal to the maximum **LONGINT** value in the implementation.

The following listing shows how to declare a **LONGINT** variable. The syntax for using **LONGINT**s is the same as for **INTEGER**s.

```
VAR MyLong : LONGINT;
```

If you try to display or use a value that is larger than **MAXLONGINT**, the result will be an overflow value. Thus, **MAXLONGINT + 1** is -2,147,483,648. Overflow handling works the same way as for **INTEGER**s.

Type UNSIGNEDWORD

CodeWarrior Pascal's **UNSIGNEDWORD** type is allocated just as much storage as an **INTEGER** (two bytes), but can only be used to represent nonnegative values. The range for **UNSIGNEDWORD** variables is thus 0 through 65535 (that is, 0 through 2^{16} - 1), inclusive. Essentially, an **UNSIGNEDWORD** differs from an **INTEGER** in that the leftmost bit is used for computing the number in an **UNSIGNEDWORD**, rather than being used as a sign bit, as for **INTEGER**s.

UNSIGNEDWORD overflow is handled just like **INTEGER** overflow, except that, in the circle, there are no negative values. Also, 65535 and 0 are adjacent.

You can tell the compiler whether you want it to do *range checking*. When this option is on, the compiler checks whether variables are being assigned values that lie outside their valid ranges. By default this option is off. You can turn this option on by checking it in the Pascal Compiler project settings.

Range checking is off by default. When this is off, the compiler will not complain if you assign **UNSIGNEDWORD** values to **INTEGER** variables, and vice versa. If the **UNSIGNEDWORD** value is greater than **MAXINT**, the overflow process will come into play. Similarly, if you assign a negative **INTEGER** value to an **UNSIGNEDWORD** variable, the result will be a different value, which uses all 16 bits to compute a magnitude.

The CodeWarrior Pascal compiler will not complain if you assign a **LONGINT** value to a two-byte variable (that is, an **INTEGER** or an **UNSIGNEDWORD**). Only two of the bytes will be assigned to the variable. For example, if you assign **MAXLONGINT** to an **UNSIGNEDWORD** variable, the **UNSIGNEDWORD** will have the value 65535. (This is because of the bit pattern in the two bytes assigned to the **UNSIGNEDWORD**.)

WARNING! It's rarely a good idea to assign LONGINT values to shorter variables or to assign UNSIGNEDWORD values to INTEGER variables.

Type UNSIGNEDLONG

CodeWarrior Pascal's **UNSIGNEDLONG** type is allocated just as much storage as a **LONGINT** (four bytes), but can only be used to represent nonnegative values. The range for **UNSIGNEDLONG** variables is thus 0 through 4,294967,295 (that is, 0 through 2^{32} - 1, inclusive.

Table 3-1. summarizes CodeWarrior Pascal's whole number types.

Table 1: CodeWarrior Pascal's Whole Number Types

Type	Size	Minimum Value	Maximum Value
INTEGER	2 bytes	-32,768	32,767
LONGINT	4 bytes	-2,147,483,648	2,147,483,647
UNSIGNEDWORD	2 bytes	0	65,535
UNSIGNED LONG	4 bytes	0	4,294,967,295

Table 3.1 Whole Number Types Defined in CodeWarrior Pascal

Character-Based Types

Pascal also allows you to represent nonnumerical information. For example, there is a predefined type for representing individual characters. In addition, CodeWarrior Pascal includes a predefined type for representing strings of characters.

The CHAR Type

Pascal's **CHAR** type is used for storing individual characters. These are stored as ASCII codes in CodeWarrior Pascal. **CHAR** variables are allocated one byte for their value, which means that the character's code can be any value from 0 through 255. (CodeWarrior Pascal's **CHAR** variables are generally allocated two bytes of storage, except when the values are to be packed, as in an array.)

To include a specific character in your program, enclose the character in single quotes. For example, the following statement assigns the letter Q to the (**CHAR**) variable **MyChar**.

```
MyChar := 'Q';
```

The standard ASCII character set (see Appendix A) contains 128 characters, with codes that range from 0 through 127. (Character codes 128 through 255 are often known as *extended ASCII* characters. You can also use such codes for special purposes in particular programs.)

In the ASCII character set, each alphabetic character is associated with a particular code. For example, *A* has ASCII code 65, and *a* has ASCII code 97. The ASCII codes for the letters are ordered and contiguous. Thus, the ASCII code for *A* is 65; for *B* it's 66; for *C* it's 67; and so on, through 90, the code for *Z*. Similarly, the lowercase letters have consecutive ASCII codes, beginning with 97 for *a*.

In addition to the upper and lower case letters, there are ASCII codes for digits (*0* through *9*), and special symbols (*#*, *%*, *&*, [, *<blank space>*, and so forth). The ASCII codes for the digits are also consecutive, beginning with 48, the code for the digit *0*. (Note that digits as characters and digits as numerical values are different. For example, `3' and 3 are different. `3' is a **CHAR** value; 3 is an **INTEGER** value. You should also be careful not to confuse the zero character (*0*) with an uppercase version of the character before *P* — namely, *O*.)

These characters (letters, digits, and special characters) are known as *printable characters*. The printable characters have the ASCII codes from 32 through 126.

To specify a printable character, put the character between single quotes, as in the preceding listing. You can also specify such characters by using the ASCII code directly, preceded by #. For example, #67 represents *C* and #106 represents *j*. When using the # representation, do *not* put quotes around the value.

In addition to the printable characters, there are *control characters*. These carry out certain actions or send particular signals. For example, TAB is a control character, as are the carriage return (CR) and linefeed (LF) characters. In fact, all the characters with ASCII codes 0 through 31 are control characters.

You can also specify control characters by using # followed by the character's code. For example, #7 specifies the character with ASCII code 7, which sends a beep.

The control characters with ASCII codes 1 through 26 each have names that include a letter. For example, ASCII code 1 is known as CTRL-A, code 2 is CTRL-B, and so on through ASCII 26, which is CTRL-Z.

The following program shows how to specify various types of **CHAR** values. Run the program a few times. What happens if you enter a character such as CTRL-D or CTRL-G?

Listing 3.18

```
(* Program to illustrate use of CHAR variables and constants *)

PROGRAM BeepTest;
CONST BeepChar = #7;
```

```
      TabChar = #9;   (* send a TAB *)
                      (* — to move cursor to next tabstop *)
VAR ChVal : CHAR;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Get a CHAR value from user *)
  PROCEDURE GetChar ( Message : STRING;
                      VAR Value : CHAR);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetChar *)

BEGIN
  GetChar ( 'Enter a Character please, then press ENTER.',
ChVal);
  WRITELN ( BeepChar, TabChar, ChVal,
            TabChar, ChVal, BeepChar);
  AwaitUser;
END.
```

In standard Pascal, the only operators available for **CHAR**s are assignment and the comparison operators. ASCII codes are used when comparing two **CHAR** values. Thus,

```
'H' < 'h'
```

is **TRUE** because the ASCII code for *H* (72) is less than the code for *h* (104).

In CodeWarrior Pascal, you can also "add" two characters. Such addition is known as *concatenation*, which is a process of putting elements together in a sequence to form another element. When you concatenate, or add, two characters, the result is a string.

Thus, the following expression assigns the value **'yz'** to the variable (**StrResult**) on the left hand side of the assignment.

```
StrResult := 'y' + 'z';  (* StrResult becomes 'yz' *)
```

The STRING *Type*

Unlike standard Pascal, CodeWarrior Pascal has a predefined **STRING** type. Variables of this type are used to store character sequences. You can either declare strings with no length specified or you can declare strings having a maximum length.

The following listing illustrates how to define three different string variables. The first variable (**DefaultStr**) has an unspecified size, or maximum length. In CodeWarrior Pascal, this means that it can have up to 255 characters. The second type (**STRING [30]**) can have at most 30 characters, and the third declaration (**STRING [75]**) specifies a string variable (**Str75**) with up to 75 characters.

```
VAR DefaultStr : STRING;    (* Maximum length = 255 chars *)
    Str30 : STRING [ 30];   (* Maximum length =  30 chars *)
    Str75 : STRING [75];    (* Maximum length =  75 chars *)
```

To specify a string value, put the character sequence within single quotes, as in the following examples:

```
DefaultStr := 'This string can have up to 255 chars';
Str30 := 'This one can have <= 30 chars';
Str75 := 'This string can have up to 75 chars';
```

If you try to assign a string that is longer than the allotted storage for the target type, the program's behavior will depend on the project settings you are using. If you set "Copy

Strings using length byte" in the Pascal Compiler Project Settings, the program will assign as many characters as possible; handling of any excess characters is undefined. Consequently, the results may be unexpected.

For example, in the following program, **ShortStr** will be assigned 30 of **RegularStr**'s characters, and the compiler will not complain about this assignment. However, the extra characters may end up overwriting the information in the next memory locations — which may contain **RegularStr**. In short, this assignment may get the original string thrashed.

On the other hand, if you turn off "Copy Strings using length byte," the compiler will flag the same assignment as an error. Thus, to get the program to compile, "Copy Strings using length byte" needs to be on. Try compiling and running the following program with this set to on.

Listing 3.19

```
(* Program to illustrate some string behaviors *)

PROGRAM StrParamDemo;
TYPE Str30 = STRING [ 30];
VAR RegularStr : STRING;
    ShortStr : Str30;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE StrParTest ( VAR NewVal, Incoming : Str30);
  BEGIN
    NewVal := Incoming;
    WRITELN ( 'IN PROCEDURE StrParTest:');
    WRITELN ( 'Incoming : ', Incoming);
    WRITELN ( 'New      : ', NewVal);
  END; (* StrParTest *)

BEGIN
```

```
  RegularStr := '1234567890123456789012345678901234567890';
  (* The following assignment may be flagged as an error
     — depending on Project settings
  *)
  ShortStr := RegularStr;

  WRITELN ( 'IN MAIN, after assigning RegularStr to ShortStr:');
  WRITELN ( 'RegularStr     : ', RegularStr);
  WRITELN ( 'ShortStr       : ', ShortStr);

  StrParTest ( ShortStr, RegularStr);
  WRITELN ( 'After calling StrParTest:');
  WRITELN ( 'RegularStr     : ', RegularStr);
  WRITELN ( 'ShortStr       : ', ShortStr);
  AwaitUser;
END.
```

Strictly speaking, the type underlying a string is not a simple type; however, you can treat it as such. This is convenient, since you'll be using strings in lots of applications throughout the book. In Chapter 7, you'll learn how strings are implemented in CodeWarrior Pascal.

As with **CHAR** values, the assignment and comparison operators are available for the **STRING** type. Again, ASCII codes are used when comparing two strings. The strings are compared character by character until the codes for corresponding characters in the two strings differ. For example,

```
'helpful' > 'hello'
```

is **TRUE** because the *p* in *helpful* has a higher ASCII code than the second *l* in *hello*. Up to that point, the two strings are identical.

You can also "add" two strings to produce a longer string. This concatenation process is the same one described for characters. The result is a string. The following listing shows this. After the assignment, **LongerStr** has the value, *a longer string*.

```
(* After the assignment, LongerStr = 'a longer string' *)
LongerStr := 'a longer' + ' string';
```

While there are few operators available for strings, there are several predefined procedures and functions available for manipulating strings. You'll learn about these in the next chapter.

Controlling Output Format for Strings

Just as with real numbers and integers, you can specify the output format for character and string variables.

To specify a field width, indicate the information to be written, write a colon (:), and then specify the field width.

If the value to be output is longer than the field width, the field width specifier is ignored. If the string value being written is shorter than the specified field width, the string is right justified.

Logical Types

In addition to numerical and character information, Pascal has a predefined type for storing logical truth values. Such values can be either true or false.

As you saw in Chapter 2, Pascal has two predefined constants, **TRUE** and **FALSE**. These two values belong to Pascal's **BOOLEAN** type. In fact, they are the *only* two values possible for **BOOLEAN** variables. (Note that there are no quotes around the Boolean values. These values are not strings; they are predefined logical values.)

A **BOOLEAN** variable is one that can have either of two values: **TRUE** or **FALSE**. The following program shows how to declare and use **BOOLEAN** variables. (The name, **BOOLEAN**, is in honor of George Boole, who developed algebraic methods for solving logical problems.)

Listing 3.20

```
(* Program to explore use of Boolean variables *)
```

```
PROGRAM ExploreBool;
USES Windows;

CONST MaxTrials = 10;
VAR Bool1, Bool2 : BOOLEAN;
    Count, Val1, Val2, RandSeed : INTEGER;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN
  (* Get and set seed value for random number generator *)
  GetInt ( 'Integer seed for random number generator', Rand-
Seed);
  SRAND ( RandSeed);
  FOR Count := 1 TO MaxTrials DO
  BEGIN
    Val1 := RAND;
    Val2 := RAND;
    IF Val1 > Val2 THEN
      Bool1 := TRUE
    ELSE
      Bool1 := FALSE;
    (* If both values are odd or both are even ... *)
    IF (Val1 - Val2) MOD 2 = 0 THEN
      Bool2 := TRUE
    ELSE
      Bool2 := FALSE;
    (* You can write the value of BOOLEAN variables *)
    WRITELN ( Val1, '  ', Val2, ';   ', Bool1, ' ', Bool2);
  END;
  AwaitUser;
END.
```

Run this program a few times, and work through a few of the results yourself. If you run the program, you'll get output such as the following.

```
21009     19020;    TRUE FALSE
14691     20014;   FALSE FALSE
28667     20844;    TRUE FALSE
 1893     26083;   FALSE  TRUE
29578     11268;    TRUE  TRUE
24378     15671;    TRUE FALSE
 4553     16152;   FALSE FALSE
17525     23100;   FALSE FALSE
28166      5061;    TRUE FALSE
24590     13053;    TRUE FALSE
Press ENTER to continue.
```

Notice that you can display the value of a **BOOLEAN** variable. You cannot, however, input the value of a **BOOLEAN** with a **READ** or **READLN** statement.

One way to assign a value to a **BOOLEAN** variable is to put a *Boolean expression* on the right side of the assignment operator. Such an expression evaluates to **TRUE** or **FALSE**.

An expression involving a comparison operator is a commonly used Boolean expression. For example, the following statements all assign (the values of) Boolean expressions to variables:

```
Bool1 := 7.5 > 13              (* assign FALSE to Bool1 *)
Bool2 := 'a' = 'a'             (* assign TRUE to Bool2 *)
Bool3 := 'z' = 'Z'             (* assign FALSE to Bool3 *)
Bool4 := TRUE <> FALSE         (* assign TRUE to Bool4 *)
Bool5 := MAXINT < MAXLONGINT   (* assign TRUE to Bool5 *)
```

Notice that a Boolean expression need not involve **BOOLEAN** operands. Of the five examples, only the fourth involves **BOOLEAN** elements. However, *all* Boolean expressions evaluate to a **BOOLEAN** value.

You'll find that Boolean expressions are crucial for defining Pascal's looping constructs (such as the **IF** and **WHILE** statements you saw in Chapter 2). You'll see much more of Boolean expressions and of **BOOLEAN** values when you learn about Pascal's control constructs.

TIP: You can display the value of a BOOLEAN; however, you cannot read the value of a BOOLEAN from the keyboard or from a file.

Operators for BOOLEAN *Variables*

You can use the assignment operator and the comparison operators with **BOOLEAN** values. When comparing two **BOOLEAN**s, you are comparing **TRUE** and **FALSE** values. For reasons that will be explained later in the chapter, **TRUE** > **FALSE**. Thus, the expression

```
TRUE > FALSE    (* evaluates to TRUE *)
```

evaluates to **TRUE**. (Note: Such a direct comparison between two **BOOLEAN** constants should never be necessary. The comparison was made to raise the issue of ordinal values and to show that even **BOOLEAN** values have ordinal values associated with them.)

In addition to these operators, there are several operators that work only with **BOOLEAN** operands. The **BOOLEAN** operators are **AND**, **OR**, and **NOT**. There are also two variants — analogous to the **AND** and **OR** operators you've just seen.

The **AND** and **&** Operators

CodeWarrior Pascal has two operators — **AND** and **&** — for testing whether two conditions are both true. The operators are equivalent in that they always give the same results for an expression. The operators differ, however, in the evaluations that actually are done.

The **AND** operator always evaluates the entire expression — that is, both left and right operands — before returning a result. In contrast, the **&** operator stops evaluating as soon as it has an unambiguous answer for the entire expression.

The **AND** operator takes two **BOOLEAN** operands and produces a **BOOLEAN** result. This operator returns **TRUE** only if *both* the left and the right operands are **TRUE**. The **&** operator gives exactly the same result.

The behavior of logical operators can be summarized in a *truth table*, which shows the outcome for every possible pairing of two **BOOLEAN** variables. Table 3-2 provides this information for the **AND** and **&** operators. The **&** operator stops evaluating after encountering the vlaue in the cells marked with an asterisk (*).

Table 2: Truth Table for AND and & Operators

Value of Left Operand (A)	Value of Right Operand (B)	Value of A AND B or of A & B
FALSE*	FALSE	FALSE
FALSE*	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

Notice that only the bottom combination results in a **TRUE**. Both operands need to be evaluated to establish the last line in the truth table. On the other hand, the top line has a result as soon as either (false) operand is evaluated. Similarly, the second line has a result as soon as the left operand is evaluated to **FALSE**; the third line has a result as soon as the *right* operand is evaluated.

In cases where it's possible, the **&** operator will stop working as soon as it has an answer. Thus, the operator only has to evaluate one operand in a false-false case. An operator that stops as soon as possible is known as a *lazy operator*. Over thousands or millions of evaluations, even the slight difference afforded by lazy operators can translate into substantial time savings.

The following program illustrates the **AND** operator:

Listing 3.21

```
(* Program to illustrate use of Boolean AND Operator *)

PROGRAM AndDemo;

CONST EvenDivisor = 6;
      OddDivisor = 7;
VAR Val1, Val2 : INTEGER;
```

```
 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Get an integer value from the user. *)
  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '?');   (* prompt user *)
    READLN ( Value);          (* read response *)
  END; (* GetInteger *)

BEGIN
  WRITELN ( ‘Testing AND Operator, by checking whether’);
  WRITELN ( ‘the values you input are divisible by’);
  WRITELN ( ‘an even and an odd divisor (6 and 7).’);
  WRITELN ( ‘——————————————‘);
  GetInteger ( 'Value 1? ', Val1);
  GetInteger ( 'Value 2? ', Val2);
  IF ( Val1 MOD EvenDivisor = 0) AND
     ( Val2 MOD OddDivisor = 0) THEN
    WRITELN ( 'Both divisors work')
  ELSE
    WRITELN ( 'Sorry, but no go.');
  AwaitUser;
END.
```

This program writes "Both divisors work" if **Val1** leaves no remainder when divided by **EvenDivisor** *and* **Val2** leaves no remainder when divided by **OddDivisor**. If either of these tests evaluates to **FALSE**, then the program writes the "Sorry, but no go" message.

Notice the parentheses around each Boolean expression. These are necessary because the **AND** operator has a higher precedence than the comparison operators. If there were no parentheses around the expressions, the **AND** operator would try to combine 0 and **Val2**. This would lead to a type conflict error, since the **AND** operator needs two **BOOLEAN** operands.

The **OR** and | Operator s

The **OR** operator also takes two **BOOLEAN** operands and produces a **BOOLEAN** result. The outcome from the **OR** operator is **FALSE** only if *both* operands are false, and **TRUE** otherwise. Thus, the **OR** operator produces a **TRUE** result if *either or both* of the operands is true. The **|** operator gives the same results, but is a lazy evaluator. Table 3-3 shows the truth table for the **OR** and **|** operators. The | operator stops evaluating after finding the values in the cells marked with and asterisk (*).

Table 3: CodeWarrior Pascal's OR and | Operators

Value of Left Operand (A)	Value of Right Operand (B)	Value of A OR B or of A | B
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE*	FALSE	TRUE
TRUE*	TRUE	TRUE

The difference between this table and the truth table for **AND** lies in the middle two cases, where the operands have different values from each other. Whereas the **AND** operator yields **FALSE** in those cases, the **OR** operator produces **TRUE**.

The following program illustrates the **OR** operator:

Listing 3.22

```
(* Program to illustrate use of Boolean OR Operator *)
```

```
PROGRAM OrDemo;

CONST EvenDivisor = 8;
      OddDivisor = 5;
VAR Val1, Val2 : INTEGER;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Get an integer value from the user. *)
  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');   (* prompt user *)
    READLN ( Value);          (* read response *)
  END; (* GetInteger *)

BEGIN
  WRITELN ( ‘Testing OR Operator, by checking whether’);
  WRITELN ( ‘the values you input are divisible by’);
  WRITELN ( ‘either an even or an odd divisor (8 and 5).’);
  WRITELN ( ‘————————————-‘);
  GetInteger ( 'Value 1? ', Val1);
  GetInteger ( 'Value 2? ', Val2);
  IF ( Val1 MOD EvenDivisor = 0) OR
     ( Val2 MOD OddDivisor = 0) THEN
    WRITELN ( 'At least one divisor works')
  ELSE
    WRITELN ( 'Sorry, but neither divisor works.');
  AwaitUser;
END.
```

This program indicates that "At least one divisor works" if either of the Boolean expressions is **TRUE** — that is, if either 8 or 5 divide their respective **Val***x* variable.

This operator also has a higher precedence than the comparison operator, so the parentheses are needed around the Boolean expressions. It's a good idea to use parentheses generously — just to be sure that you and the compiler interpret statements the same way.

The **XOR** Operator

The preceding **BOOLEAN** operators are defined in standard Pascal. In addition, CodeWarrior Pascal has an **XOR** operator, which yields **TRUE** if exactly one of the two operands is **TRUE**. For this operator, *only* the mixed cases yield true, as you can see in Table 3-4.

Table 4: CodeWarrior Pascal's XOR Operator

Value of Left Operand (A)	Value of Right Operand (B)	Value of A XOR B
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE

This operator differs from the **OR** operator only in the last line of the truth table. Whereas the **OR** operator allows both operands to be **TRUE**, the **XOR** (for e**X**clusive **OR**) will not allow both operands to be **TRUE**.

The following program illustrates how to use the **XOR**, as well as the other two Boolean operators you've seen so far:

Listing 3.23

```
(* Program to illustrate use of Boolean XOR Operator *)

PROGRAM BoolOpsDemo;

CONST EvenDivisor = 6;
      OddDivisor = 7;
      MaxCount = 50;
VAR Count : INTEGER;

 (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  WRITELN ( 'AND' : 10, 'OR' : 10, 'XOR' : 10);
  WRITELN;
  FOR Count := 1 TO MaxCount DO
  BEGIN
    IF ( Count MOD EvenDivisor = 0) AND
       ( Count MOD OddDivisor = 0) THEN
      WRITELN ( Count : 10);
    IF ( Count MOD EvenDivisor = 0) OR
       ( Count MOD OddDivisor = 0) THEN
      WRITELN ( Count : 20);
    IF ( Count MOD EvenDivisor = 0) XOR
       ( Count MOD OddDivisor = 0) THEN
      WRITELN ( Count : 30);
  END; (* FOR Count ... *)
  AwaitUser;
END.
```

This program displays values between 1 and **MaxCount** that have certain properties — as tested in the Boolean expressions.

The **NOT** Operator

The remaining Boolean operator is a unary operator. The **NOT** operator takes a **BOOLEAN** operand and yields a **BOOLEAN** whose value is the opposite of the operand's value. Thus, if the operand is **TRUE**, the **NOT** operator yields **FALSE**.

You can apply the **NOT** operator to any **BOOLEAN** value or variable or to a Boolean expression. The first five statements in the following listing are valid applications of the **NOT** operator, the remaining three are invalid:

```
(* valid *)
NOT NOT FALSE                 (* returns FALSE *)
NOT TRUE                      (* returns FALSE *)
NOT ( 3 > 5)                  (* returns TRUE *)
NOT (( 3 > 5) AND ( 5 < 7))   (* returns TRUE *)
NOT ( 3 < 5) AND ( 5 < 7)     (* returns FALSE *)

(* invalid *)
NOT ( 3)            (* can't use NOT with INTEGER *)
NOT ( 'g')          (* can't use NOT with CHAR *)
NOT ( 'goodbye')    (* can't use NOT with STRING *)
```

Notice that you don't need parentheses around the operand for the **NOT** operator. Also, compare the last two valid statements. These differ in that the fourth has parentheses around the entire Boolean expression involving the **AND** operator.

The two statements produce different results because of the relative precedence of the **NOT** and the **AND** operators. The former has higher precedence. Therefore, in the last statement, the expression

```
NOT ( 3 < 5)
```

is evaluated (to **FALSE**) and this result becomes the left operand for the **AND** operator.

The following program illustrates the **NOT** operator:

Listing 3.24

```
(* Program to explore use of Boolean NOT operator *)

PROGRAM ExploreNot;
USES Windows;

CONST MaxTrials = 10;
VAR Bool1, Bool2 : BOOLEAN;
    Count, Val1, Val2, RandSeed : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN
  (* Get and set seed value for random number generator *)
  GetInt ( 'Integer seed for random number generator', RandSeed);
  SRAND ( RandSeed);
  FOR Count := 1 TO MaxTrials DO
  BEGIN
    Val1 := RAND;
    Val2 := RAND;
    (* The following Boolean expression is TRUE if
       Val1 is LESS than Val2
    *)
    IF NOT ( Val1 > Val2) THEN
      Bool1 := TRUE
    ELSE
```

```
      Bool1 := FALSE;
    (* If both values are odd or both are even ... *)
    IF (Val1 - Val2) MOD 2 = 0 THEN
      Bool2 := TRUE
    ELSE
      Bool2 := FALSE;
    (* You can write the value of BOOLEAN variables *)
    WRITELN ( Bool1, ' ', Bool2);
  END;
  AwaitUser;
END.
```

The **NOT** operator can also be used to change the values of individual bits in the representation of a whole number type. You'll learn about the *bit-handling operators* in Chapter 10.

Precedence for the BOOLEAN *Operators*

All of the Boolean operators have a higher precedence than the comparison operators. Table 3-5 shows the order of precedence for all of the operators you've seen so far, including the Boolean operators.

Table 5: Precedence of CodeWarrior Pascal's Operators

Operators	Comments
NOT	BOOLEAN Negation
*, **, /, DIV, MOD, AND, &	Multiplicative Operators
+, -, OR, |, XOR	Additive Operators
=, <>, >, >=, <. <=	Comparison Operators
:=	Assignment Operator

Enumerated Types

Pascal enables you to create your own simple type, just by listing, or enumerating, the possible values this type can take. Once you've done this, you can declare variables of this type. Because you list, or enumerate, the individual values, such types are known as *enumerated* types.

For example, suppose you want to represent a deck of cards. To represent a card, you need the card's suit and its face value. The following program shows how to create variables to represent both of these items of information — without resorting to arbitrary numerical codes for these values. Note that the program doesn't do anything observable; it simply shows the syntax for assigning enumerated values to variables. Once you've learned more about Pascal's syntax, you may want to expand this program — for example, to deal poker or gin rummy hands.

Listing 3.25

```
(* Program illustrating how to define, declare and
   use enumerated types.
*)
PROGRAM Cards;

(* Define a type by listing the possible values
   variables of this type can take.
*)
TYPE CardSuit = ( club, diamond, heart, spade);
     FaceValue = ( two, three, four, five, six, seven, eight,
                   nine, ten, jack, queen, king, ace);

VAR Suit : CardSuit;
    Face : FaceValue;
    Ch : CHAR;

BEGIN
  Suit := club;
  Face := ten;
  WRITELN ( 'Just specified values for a 10 of clubs.');
  WRITELN ( 'Press ENTER to end program. ');
  READLN ( Ch);
END.
```

The **Cards** program defines the appropriate variables, and then makes the assignments for a ten of clubs.

Using a TYPE Definition to Create the Type

The first step in working with enumerated types is to create a **TYPE** definition. You must do this in the declaration section, and must specify a name for the type and a list of the values that the type can have. This list of values appears in parentheses, with individual values separated by commas.

The equal sign and the parentheses are fixed; the remaining elements will vary with each definition. Notice that the type values are not within quotes. These values are not strings; rather, they are names to be associated with internal codes.

Internally, CodeWarrior Pascal simply associates the numerical codes 0 through <number of values - 1> with the values you've listed. This makes it easy for the compiler to use your new type; the meaningful value names make it easier for you to read the program.

One consequence of using such a representation is that values are actually ordered. For example,

```
hearts > club
```

because the internal code (2) for **heart** is larger than the code (0) for **club**. Similarly, an **ace** value is higher than a **jack**.

This ordering may be useful in a program. For example, to compare two cards, you could use a comparison operator to determine which value was "larger." The assignment and the comparison operators are the only operators available for enumerated types.

You can define multiple types. The first type definition line begins with the word **TYPE**; subsequent type definition lines begin with an identifier or with blank space.

The example program defined two enumerated types: **CardSuit** and **FaceValue**. The first of these can take on four different values, the second can take on 13.

A **TYPE** definition is simply a description. Such a definition does not create any variables of the specified type, nor is any storage allocated to store variables of that type.

The amount of storage allocated for such variables depends on the possible values the type can have. For types that have fewer than 256 different values, one byte is allocated; for types with more values, two bytes are allocated.

Declaring Enumerated Variables

No **CardSuit** variables exist until the declaration of **Suit**; similarly, no storage is allocated for any **FaceValue** variables until **Face** is declared.

Once you've defined a type, you can declare variables of this type. To do so, use the same syntax as for other types, as in the variable declarations. In the examples, two separate variables are used to represent the suit and face value information. In Chapter 7, you'll learn about a more complex type that you can use to represent both of these items of information in the same variable.

Using Enumerated Variables

Once you have declared the appropriate variables, you can do what you wish. The assignment operator and the comparison operators are the only ones that work with enumerated types. They are the only operators that work with such types.

Assignments to enumerated variables have the same format as other assignments.

Note that the enumerated type values are not within quotes in the assignment statements. Again, this is because these are not strings.

You'll find lots of uses for enumerated types in your programs. Since you can use meaningful names for the values (instead of using arbitrary numerical codes) it is easier to keep an overview of your actions with such variables.

Unfortunately, you can't write the value of an enumerated type variable directly to the screen or a file. Nor can you read the value of such a variable directly.

Summary

This chapter covered CodeWarrior Pascal's simple data types. As you saw, you've got lots of types to choose from, and you can even create your own if you need to.

In the next chapter, you'll learn about some of the predefined procedures and functions available for use with these simple types. You'll also start learning how to define your own procedures and functions.

4

Procedures & Functions

In the last chapter, you learned about Pascal's simple types. You also learned about various components of a Pascal program, such as statements, comments, constant definitions, and variable declarations. In this chapter, you'll start learning about two other important program elements: procedures and functions.

Whereas statements (individual instructions) are the fundamental elements of Pascal programs, a program is really built out of procedures and functions, which consist of statements. The main program calls these procedures and functions to do its work. (To simplify the discussion, we'll use *routines* to refer to both procedures and functions.)

Before discussing how to define your own routines, let's look briefly at what procedures are and how they work. You'll also learn how Pascal's functions differ from procedures.

Procedures: General Features

A *procedure* is a collection of statements grouped together to accomplish a particular task. The procedure's name refers to this collection of statements.

In many ways, a procedure is really a miniature program. As you'll find, a procedure can have its own declaration section and procedure body.

Procedures are designed to carry out some task, such as clearing the screen, sorting a collection of values, beeping at the user, and so forth. Sometimes a procedure will need to perform a general task with variations each time. For example, a procedure may add a collection of numbers. Each time the procedure is called, the values and the number of values to add will be different. To work properly, the procedure will need to be able to deal with such differences.

To determine what the variants are, procedures need to be able to communicate with the caller (the main program or another routine). Procedures can communicate with callers in two ways:

- By using global variables, which are accessible to other routines
- By using parameters to pass information into and out of the procedure.

We'll look at these alternatives later in the chapter.

Procedure Definitions and Calls

It's important to distinguish between procedure definitions and procedure calls. To use a procedure, you just need to call it in your program. Before you can use a procedure, however, you need to define it.

A *procedure definition* specifies the variables used by the procedure, and also specifies the sequence of instructions to be executed in accomplishing the procedure's task. Such a definition appears in the declaration section of your program or, as you 'll see in Chapter 6, of another routine.

Once you've defined a procedure, you can use it in your program. To accomplish the task for which the procedure was defined — you simply need to call the procedure. A *procedure call* is a mechanism by which program control is passed temporarily to the called procedure.

TIP: In Pascal, you need to define a procedure before you can call it.

When a procedure is called, the instructions contained in the procedure definition are carried out. After the procedure finishes executing, control reverts back to a point in the calling routine that is immediately after the procedure call.

The following listing shows how this process works.

Listing 4.1

```
PROGRAM ProcedureShell;

VAR Val1, Val2 : INTEGER;
    Ch : CHAR;
```

```
                 PROCEDURE ChangeInfo;
                 BEGIN
(* 10 *)           WRITELN ( ‘Beginning ChangeInfo.’);
(* 11 *)           Val1 := Val1 * Val2;
(* 12 *)           Val2 := Val1 + Val2;
(* 13 *)           Val1 := Val1 * Val2;
(* 14 *)           WRITELN ( ‘Ending ChangeInfo.’);
                 END;  (* ChangeInfo *)
BEGIN
(* 1 *)  Val1 := 1;
(* 2 *)  Val2 := 2;
(* 3 *)  WRITELN ( ‘Calling ChangeInfo.’);
(* 4 *)  ChangeInfo;
(* 5 *)  WRITELN ( ‘Back in Main, from ChangeInfo.’);
(* 6 *)  WRITELN ( Val1);
(* 7 *)  WRITELN ( Val2);
(* 8 *)  WRITELN ( 'Press ENTER to end program');
(* 9 *)  READLN (Ch);
END.
```

The flow of control in the program is as follows:

- The program starts executing at the top of the program body — line 1 in the example. The assignments in lines 1 and 2 are carried out.
-
- In line 3, the program tells you it’s about to transfer control to **ChangeInfo**.
- When the main program reaches line 4, it stops executing temporarily.
- Control is transferred to procedure **ChangeInfo**. The next statement to execute will be the announcement in line 10, followed by the assignments in lines 11-13, and a second announcement in line 14..
- The statements in **ChangeInfo** are executed in sequence — that is, lines 10, 11, 12, 13, and 14.

- After line 14 has finished executing, control transfers back to the main program — continuing at line 5.
- The main program resumes execution with the main program's announcement in line 5.
- Lines 5, 6, and 7 are executed in sequence.
- The program lets the user officially end the program in lines 8 and 9.

The calls to **WRITELN** have the same results as the call to **ChangeInfo**. That is, the main program is suspended temporarily, while procedure **WRITELN** executes. Although **WRITELN** is predefined, it isn't very different from procedures that you write.

The **ProcedureShell** example program provides the simplest scenario of how the flow of control changes as the program executes. In the example, all of the procedure calls are made from the main program. However, procedures can also call each other, which means that the flow of control through the program can get quite involved.

For example, one of the statements in **ChangeInfo** might have been a call to another procedure (let's say, **VerifyChanges**). At that call, **ChangeInfo** stops executing temporarily, while control is transferred to **VerifyChanges**.

At this point, both the main program *and* procedure **ChangeInfo** are suspended. **VerifyChanges** executes its statements until it finishes its task. When **VerifyChanges** is finished, control is returned to **ChangeInfo**, which resumes executing with *its* next statement.

Of course, **VerifyChanges** may have called **MakeMoreChanges**, in which case, ..., well, you get the picture.

When a procedure is called, a temporary environment is set up for the procedure. This environment includes temporary storage, which is allocated for any variables that are declared in the procedure. This environment and storage will “disappear" once the procedure finishes.

That is, the memory locations allocated for the procedure's variables will be returned to a common pool of available memory for use by the program. One consequence of this is that you won't be able to access the storage allocated for a procedure's variable once you leave the procedure.

Pascal Procedures

With this general discussion of procedures to provide a context, let's look at how procedures are defined and used in CodeWarrior Pascal. In Pascal, the structure of a procedure is very similar to that of a program. In particular, a procedure consists of

- A procedure heading
- An optional declaration section
- A procedure body

The Procedure Heading

The procedure heading — also known as the *interface* — consists of the reserved word **PROCEDURE**, followed by a name for the procedure. The procedure name can be any valid identifier, and may be followed by a parameter list (discussed shortly). The heading will be followed by a semicolon to separate it from subsequent source code.

The following are all examples of procedure headings. The first two have no parameter lists, and the rest do. **GetDate**, the last procedure in the example listing, has three parameters, all of the same type.

```
PROCEDURE BeepUser;

PROCEDURE Wait;

PROCEDURE Count ( HowHigh : INTEGER);

PROCEDURE BubbleSort ( VAR Info : String80Array;
                           Size : INTEGER);

PROCEDURE GetString ( Message : STRING;
                      VAR Value : STRING);

PROCEDURE GetDate ( VAR Month, Date, Year : INTEGER);
```

The Declaration Section

Procedures can have their own declaration sections — in which you can define constants and types, and you can declare variables. You can even define other procedures and functions in a procedure's declaration section.

The syntax for these definitions and declarations is the same as for the program's declaration section. There are differences, however, in the accessibility of the objects defined and declared in this declaration section, as you'll see later.

The Procedure Body

The procedure body contains the statements that accomplish the task for which the procedure was defined. The procedure body starts with **BEGIN** and terminates with **END**. Unlike the **END** that terminates the program body, however, the **END** after a procedure body is followed by a semicolon.

When control transfers to a procedure, it begins executing with the first statement in the procedure body and continues executing until the last statement has been carried out. After the last statement, control returns to the calling routine.

TIP: The END that terminates a procedure body is followed by a semicolon, not a period.

Global and Local Variables

Earlier, you learned that a procedure could communicate with a caller using either global variables or parameters. In this section, you'll learn about global variables and also about their counterpart, local variables.

A *global variable* is one that is accessible to the main program and to routines defined within the main program. In Chapter 3, you learned how to declare variables in the program body. Such declarations allocate memory for global variables.

For now, think of a global variable as one that is defined in the main program. You can also declare variables in the declaration section for a procedure or function, but those will not be global variables. Rather, such declarations are known as *local variables*.

For example, the following program contains several procedures with local declarations. Notice that you can specify constants as well as declare variables. In fact, you can even create types and define routines.

Listing 4.2

```
(* Program to illustrate declaration and use of
   local and global variables.
*)

PROGRAM LocalDemo;
USES Windows;

   (* Global Declarations *)
  CONST MaxTrials = 10000;
  VAR Qtile1, Qtile2, Qtile3, Qtile4, RandSeed : INTEGER;
      Ch : CHAR;

   (* Generate random values between 0.0 and 1.0,
      classifying result into one of four groups.
      Procedure contains local declarations,
      but also uses several global variables.
  *)
  PROCEDURE RandomSamples;
   (* Specify 3 local constants and 2 local variables *)
  CONST Cutoff1 = 0.25;
        Cutoff2 = 0.50;
        Cutoff3 = 0.75;
  VAR   Result : REAL;
        Count : INTEGER;
  BEGIN
    WRITELN ( '  Starting RandomSamples');
     (* MaxTrials is a global constant *)
    WRITELN ( 'Cutoff1 = ', Cutoff1, '; cutoff2 = ', Cutoff2,
               '; cutoff3 = ', Cutoff3);
    FOR Count := 1 TO MaxTrials DO
    BEGIN
      Result := RAND / MAXINT;
      IF Count <= 10  THEN
        WRITELN ( Count, ' : ', Result);
       (* Classify resulting value into the following groups:
          0.0 <= value < Cutoff1 (0.25)
          Cutoff1 <= value < Cutoff2 (0.50)
          Cutoff2 <= value < Cutoff3 (0.75)
        Cutoff3 <= value < 1.0.
        The QtileX variables are all global.
     *)
     IF Result < Cutoff1 THEN
       QTile1 := QTile1 + 1
     ELSE IF Result < Cutoff2 THEN
       QTile2 := QTile2 + 1
     ELSE IF Result < Cutoff3 THEN
       QTile3 := QTile3 + 1
     ELSE
       QTile4 := QTile4 + 1;
   END; (* FOR Count := 1 TO MaxTrials *)
   WRITELN ( '  Ending RandomSamples');
 END; (* RandomSamples *)

 (* Display values of four global variables.
    Procedure also shows that you can use identifiers
    to specify output format.
 *)
 PROCEDURE DisplayResults;
 CONST FWidth = 8;
       Precision = 4;
 BEGIN
   WRITELN ( '  Starting DisplayResults');
   WRITELN ( '  Qtile1 = ', Qtile1 : FWidth);
   WRITELN ( '  Qtile2 = ', Qtile2 : FWidth);
   WRITELN ( '  Qtile3 = ', Qtile3 : FWidth);
   WRITELN ( '  Qtile4 = ', Qtile4 : FWidth);
   WRITELN ( '  Ending DisplayResults');
 END; (* DisplayResults *)

 (* Write a Beep Character *)
 PROCEDURE BeepUser;
 CONST BeepChar = #7; (* ASCII 7 *)
 BEGIN
   WRITELN ( '  Starting BeepUser');
   WRITE ( BeepChar);
   WRITELN ( '  Ending BeepUser');
 END;  (* BeepUser *)
```

```
  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN
  WRITELN ( 'Starting Main program');
  (* Get and set seed value for random number generator *)
  GetInt ( 'Integer seed for random number generator', RandSeed);
  SRAND ( RandSeed);
  Qtile1 := 0;
  Qtile2 := 0;
  Qtile3 := 0;
  Qtile4 := 0;
  RandomSamples;
  DisplayResults;
  BeepUser;
  WRITELN ( 'Press ENTER to end Main program');
  READLN ( Ch);
END.
```

This program generates 10,000 random values between 0.0 and 1.0. Depending on the value of each trial, a counter for one of four groups is incremented. After generating these values, a second procedure displays the results of the sampling. Finally, the program beeps and ends.

The **LocalDemo** program contains both global and local variables. The declarations and constant specifications in the main program (four **REAL**s and one **CONST**) represent the global variables. The declarations within each of the procedures represent local variables and constants.

What's the difference between local and global variables? For one thing, global variables can be used anywhere in the program. Thus, both **RandomSamples** and **DisplayResults** use the global **Qtile***X* variables.

Local variables, on the other hand, can be used, and are recognized, only in the procedure in which the variables were declared. Essentially, a procedure creates a miniature environment for its declarations and definitions, and for the procedure body. Local variables are a part of this environment, and cannot exist outside of it.

For example, the constants **FWidth** and **Precision** are recognized only within **DisplayResults**. If you try to refer to these constants in the main program or in one of the other procedures, you'll get an “undeclared identifier" compiler error. This is because these constants don't exist when **DisplayResults** is not activated. Similarly, only **RandomSamples** can use **Result** in this program.

In Chapter 6, you'll learn how to define routines within other routines. There's also more to local and global environments, as you'll see in that chapter.

Parameters

Throughout the early chapters you've been calling procedures and passing arguments to these procedures. The arguments provide information that the procedures need to carry out their tasks. For example, a sorting procedure will need to know what values you want to sort, and may need to know how many values are to be sorted. You can pass this information to the procedure as arguments.

How do you *define* a procedure that takes an argument when called? Earlier you learned that a procedure heading might have a parameter list. This is where you would prepare the procedure's environment for the information that will be passed as arguments.

A *parameter* is a slot through which information can be passed. In Pascal, such a slot will be of a particular size — depending on the type of information being passed. Parameters can represent either one-way or two-way slots for information transfer.

When you define a procedure, you'll have to provide slots for every argument that will be passed to the procedure in a call. For example, the procedure in the following program counts to a number specified in an argument. Look at the definition of **CountVals** to see how the parameter is handled.

Listing 4.3

```
PROGRAM ProcExample;

VAR NrTimes : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE CountVals ( HowHigh : INTEGER);
  VAR Index : INTEGER;
  BEGIN
    FOR Index := 1 TO HowHigh DO
      WRITELN ( Index);
  END; (* CountVals *)

BEGIN
  CountVals ( 30);
  NrTimes := 95;
  CountVals ( NrTimes);
  AwaitUser;
END.
```

Procedure **CountVals** takes one argument: the value to which the procedure should count. When you call **CountVals**, this value is specified within the parentheses after the procedure name. In the two calls to **CountVals**, the arguments are 30 and **NrTimes** (which evaluates to 95), respectively.

When defining the **CountVals** procedure, you need to indicate that this procedure will get an item of information when called. You do this by including a *formal parameter* declaration within the parentheses in the procedure heading. This parameter declaration specifies a name and a type for each slot through which information is being passed to or from the procedure.

Multiple Parameters

You can declare multiple parameters for a procedure. These parameters can be of the same or different types. The following listing contains several procedure headings, to illustrate the various aspects of procedure declaration syntax.

```
PROCEDURE CountVals ( HowHigh : INTEGER);

PROCEDURE BubbleSort ( VAR Info : String80Array;
                           Size : INTEGER);

PROCEDURE GetString ( Message : STRING;
                      VAR Value : STRING);

PROCEDURE GetDate ( VAR Month, Date, Year : INTEGER);
```

The identifier may be preceded by the reserved word **VAR**.

If you have multiple parameters of the same type, you can list them all as part of the same declaration. You need to separate the individual parameter names with commas, as in the interface for **GetDate** in the preceding listing.

If you have multiple parameters of *different* types, the individual parameter declarations for the types must be separated by a semicolon, as in the headings for **BubbleSort** and **GetString**. The parameters need not be listed on separate lines, although such a style makes them easier to read.

Formal Parameters and Arguments

When you specify formal parameters in a procedure heading, you're defining slots through which information will be passed for use by the procedure. When you *call* the procedure, you put information into these slots by specifying, or *passing*, arguments. Whereas formal parameters represent slots, arguments represent actual information. (In fact, arguments are often known as *actual parameters*.)

For each formal parameter you specify, you *must* pass an argument when you call the procedure. Thus, you must specify two arguments when calling **GetInteger**, whereas you only need to specify one when calling **CountVals**.

You also need to pass the arguments in the correct order. Each formal parameter will have a type associated with it, and the formal parameters will be specified in some order. Each actual parameter must also have a type associated with it, and the type for each argument must match the type for the corresponding formal parameter. (Certain predefined routines are exceptions to these restrictions. For example, **WRITELN** can take a variable number of arguments, and these may be of different types each time the procedure is called.)

The first five calls in the following listing are valid, but the next four are not:

```
                 (* Valid Calls *)
(* Assume MyInt is an initialized INTEGER variable *)
(* Assume MyStr is an initialized STRING variable *)
CountVals ( MyInt);   (* can pass a variable as a parameter *)
CountVals ( 30);       (* can pass an integer value *)
CountVals ( 2 * 15);   (* can pass an integer expression *)

(* After each of the following calls,
   MyInt will have a new value.
*)
GetInteger ( MyStrVal, MyInt);  (* STRING variable is OK *)
GetInteger ( `Value?', MyInt);  (* STRING constant is OK *)

                 (* Invalid Calls *)
CountVals ( MyInt, 30);        (* too many arguments *)
CountVals ( 20.3);            (* argument must be INTEGER *)
GetInteger ( MyInt);          (* first argument's missing *)
GetInteger ( MyInt, 'Value?'); (* arguments are in
                                  the wrong order *)
```

Declaring and Using Parameters

What happens when you declare a parameter for a procedure? Such a declaration is essentially an instruction to allocate storage whenever the procedure is called. The storage should be of the size required to represent a value of the type specified for the parameter.

For example, when you declare **HowHigh** as a formal parameter for **CountVals**, two bytes of temporary storage will be allocated whenever **CountVals** is called. This storage will be freed for reuse when the procedure has finished.

What happens when you pass an argument in a procedure call? When you call **CountVals** with an argument of 30, the program copies this value into the storage allocated temporarily for **HowHigh**. Similarly, when the program calls **CountVals** with **NrTimes** as the argument, the current value of **NrTimes** is copied to **HowHigh** — that is, to the storage allocated for **HowHigh**. Once this value is copied, nothing further is done with **NrTimes**. In this case, the value of **HowHigh** does not change as a result of the procedure call.

When calling a procedure such as **CountVals** with an argument, only the argument's value is of interest; the source of that value is not. Because only values count, such parameters are said to be *passed by value*. Shortly, you'll learn about an alternative way of passing parameters.

Because the value of such an argument is copied to the procedure's local environment, you'd better be certain your argument *has* a value — that is, that it has at least been initialized. Otherwise, the procedure will work with gibberish, and will give you incorrect results.

You can pass information by value in various forms:

as a constant value (for example, 30 or `A string constant'),

- as a variable (for example, **NrTimes** or **GreetingStr**)
- as an expression (for example, 3 + 5 or 7 < 5)

When you declare a parameter, temporary storage will be allocated for the parameter. In addition, the parameter's name is associated with the allocated storage. This name was specified within the procedure's environment (in particular, in the parameter list for the procedure) and is, therefore, a local name.

Thus, when procedure **GetInteger** is executing, there will be areas of memory associated with the identifiers **Message** and **Value**. These names will *not* be recognized outside the procedure's environment.

Similarly, the storage allocated for an **INTEGER** in **CountVal**'s local environment is known as **HowHigh** while **CountVal** is executing. Once the procedure is finished, the name — along with the storage — disappears. Like the storage, the name is local to the procedure: The main program does not know that the identifier **HowHigh** exists.

The calling environment does not know the local names for the parameters. It knows only the number and kinds of slots that will need to be passed information. For example, when **GetInteger** is called, whatever value is in the first slot will be referred to as **Message**, and whatever value is in the second slot will be referred to as **Value**.

To the calling environment, the information in such slots has different names. For example, in the main program the first argument might be a string variable (such as **StrVariable**) or constant, and the second might be an **INTEGER** variable (such as **MyInt**).

Two-Way Parameters

You've been using parameters to pass information *into* your procedures. The procedure may need an item of information to do its work, but this information is not changed for the calling routine. However, sometimes you'll need to pass information back out from a procedure to the calling routine. For example, procedure **GetInteger** needs to provide an **INTEGER** value to the calling routine. In the case of **GetInteger**, this is accomplished by returning the desired value in the second parameter. **GetInteger**'s second parameter declaration begins with the reserved word **VAR**. This word indicates that the parameter (**Value**) will be used differently than the **Message** parameter.

A parameter whose declaration begins with **VAR** is intended for passing information back *from* a procedure to the calling routine. When the procedure finishes executing the **VAR** parameter will have a particular value. The argument that was originally passed from the calling routine to the procedure will have this same new value. (As it turns out, this is because both identifiers — argument and formal parameter names — refer to the same memory location.)

In the following example, you can examine two-way parameter passing more closely. The **ParamTest** program contains two versions of the same procedure. Each version is supposed to swap the values in two variables.

Listing 4.4

```
(* Program to illustrate two different uses of parameters *)

PROGRAM ParamTest;

VAR Val1, Val2 : INTEGER;

  (* Ostensibly swaps contents of First and Second;
     DOES NOT DO ITS TASK.
     Works with copies, so the changes disappear
     when the procedure finishes executing.
  *)
  PROCEDURE BadIntSwap ( First : INTEGER;
                         Second : INTEGER);
  VAR Temp : INTEGER;
  BEGIN
     Temp := First;
     First := Second;
     Second := Temp;
  END; (* BadIntSwap *)

  (* Correctly swaps contents of First and Second;
     Procedure works with storage allocated for arguments, so
     the changes are made to the variables passed as arguments.
     This means the changes endure once the procedure finishes.
  *)
  PROCEDURE GoodIntSwap ( VAR First : INTEGER;
                          VAR Second : INTEGER);
  VAR Temp : INTEGER;
  BEGIN
     Temp := First;
     First := Second;
     Second := Temp;
  END; (* GooodIntSwap *)

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN  (* Main program *)
  GetInteger ( 'Value 1?', Val1);
  GetInteger ( 'Value 2?', Val2);
  WRITELN ( 'Original Val1 = ', Val1:5, ';  Val2 = ', Val2:5);
  BadIntSwap ( Val1, Val2);
  WRITELN ( 'After BadIntSwap');
  (* The following line will be identical to first display *)
  WRITELN ( '         Val1 = ', Val1:5, ';  Val2 = ', Val2:5);
  GoodIntSwap ( Val1, Val2);
  WRITELN ( 'After GoodIntSwap');
  WRITELN ( '         Val1 = ', Val1:5, ';  Val2 = ', Val2:5);
  AwaitUser;
END.
```

Run the program and compare the output from the two versions. This program produces the following output, for inputs of 12 and 13.

```
Original Val1 =    12;  Val2 =    13
After BadSwap
         Val1 =    12;  Val2 =    13
After GoodSwap
         Val1 =    13;  Val2 =    12
```

First, notice that nothing changed when **BadIntSwap** did its work; the two values *were* exchanged by **GoodIntSwap**, however. The interfaces for the two procedures indicate that the difference lies in the **VAR**s preceding the parameter names in **GoodIntSwap**.

There were no changes in the values of **Val1** and **Val2** after they had been passed to **BadIntSwap** — because **BadIntSwap** works only with copied information. When the arguments are passed to **BadIntSwap**, the arguments' values are copied to **First** and **Second**. Since only copies were involved in **BadIntSwap**, it should come as no surprise that the values were not exchanged in the main program.

In contrast, when **GoodIntSwap** finishes, the values of the two parameters — **Val1** and **Val2** in the main program — have been exchanged. Somehow, the procedure knew where the two arguments were located — to make the exchanges.

Essentially, procedure **GoodIntSwap** had access to variables **Val1** and **Val2** — even though these names are not even mentioned in the procedure's body. The procedure refers to the memory locations at which it does its work as **First** and **Second**. These locations correspond to the locations of **Val1** and **Val2**. This means that changes to **First** and **Second** are also changes to **Val1** and **Val2**.

Based on the changes in their values after **GoodIntSwap** has finished, you might think that **Val1** and **Val2** were passed directly — rather than in the form of copies. Such parameters are said to be *passed by reference*.

Parameters that are passed by reference *can* be changed in the called procedure. Rather than allocating temporary storage for such parameters when the procedure is called, the compiler allocates enough storage to represent the location of the variables in the calling routine.

Because parameters that are passed by reference must be associated with a memory location (in which the changes will be made), you cannot pass specific values (such as 30) or expressions (such as 3 + 5) to such parameters. You can only pass variables by reference — so that a memory location known to the calling routine can be changed.

You'll learn more about local and global names and variables, and about parameter passing in Chapter 6. For now, let's look at another example, to illustrate the difference between passing by value and by reference.

Examples of Parameter Passing

The following program calls three procedures which are identical except for their parameter lists. Each routine has two parameters, but each routine differs from the others in the number of **VAR** parameters.

For example, **AllValues** has no **VAR** parameters, so both arguments are passed by value. This means that copies of the arguments are passed to the procedure. None of the changes to these copies will affect the original arguments.

ValueVar has one parameter (the first) that is passed by value, and a second that is passed by reference. In this case, one of the parameters is associated with storage accessible through the calling routine. Changes made at this location will remain in effect, even once **ValueVar** has finished executing.

Finally, for **AllVars** both parameters are passed by reference. This means that both of the parameters will affect values in the calling routine.

Listing 4.5

```
(* Program to illustrate passing by value and by reference.
   Program also shows that same identifiers can be used
   in multiple places in a program.
*)

PROGRAM ParamDemo;
VAR G1, G2 : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)
```

```
(* Both arguments are passed by value, so no changes
   endure after procedure finishes.
*)
PROCEDURE AllValues ( X, Y: INTEGER);
BEGIN
  WRITELN ( '    Entering AllValues   :  X = ', X,
            ';  Y = ', Y);
  X := X * 2;
  Y := Y * 4;
  WRITELN ( '    Leaving AllValues    :  X = ', X,
            ';  Y = ', Y);
END;  (*  AllValues .. 2 *)

(* First argument is passed by value; second by reference.
   Only second parameter's value remains — as the new
   value for the second argument when ValuVar was called.
*)
PROCEDURE ValueVar ( X : INTEGER; VAR Y: INTEGER);
BEGIN
  WRITELN ( '    Entering ValueVar    :  X = ', X,
            ';  Y = ', Y);
  X := X * 2;
  Y := Y * 4;
  WRITELN ( '    Leaving ValueVar     :  X = ', X,
            ';  Y = ', Y);
END;  (*  ValueVar .. 2 *)

(* Both arguments are passed by reference,
   so both arguments passed in the call to AllVars
   will have new values.
*)
PROCEDURE AllVars ( VAR X, Y: INTEGER);
BEGIN
  WRITELN ( '    Entering AllVars     :  X = ', X,
            ';  Y = ', Y);
  X := X * 2;
  Y := Y * 4;
  WRITELN ( '    Leaving AllVars      :  X = ', X,
            ';  Y = ', Y);
  END;  (*  AllVars .. 2 *)

BEGIN (* Main Program *)
  G1 := 10;     G2 := 20;
  WRITELN ( 'Calling AllValues        : G1 = ', G1,
            '; G2 = ', G2);
  AllValues ( G1, G2);
  WRITELN ( 'Returning from AllValues : G1 = ', G1,
            '; G2 = ', G2);
  WRITELN;

  WRITELN ( 'Calling ValueVar         : G1 = ', G1,
            '; G2 = ', G2);
  ValueVar ( G1, G2);
  WRITELN ( 'Returning from ValueVar  : G1 = ', G1,
            '; G2 = ', G2);
  WRITELN;

  WRITELN ( 'Calling AllVars          : G1 = ', G1,
            '; G2 = ', G2);
  AllVars ( G1, G2);
  WRITELN ( 'Returning from AllVars   : G1 = ', G1,
            '; G2 = ', G2);
  AwaitUser;
END.
```

The following listing shows the output from this program. **X** and **Y** are the local names for **G1** and **G2**, respectively. Compare the values while in each routine with the values in the main program when the routine has finished executing.

```
Calling AllValues        : G1 = 10; G2 = 20
    Entering AllValues   :  X = 10;  Y = 20
    Leaving AllValues    :  X = 20;  Y = 80
Returning from AllValues : G1 = 10; G2 = 20
```

```
Calling ValueVar        : G1 = 10; G2 = 20
    Entering ValueVar   :  X = 10;  Y = 20
    Leaving ValueVar    :  X = 20;  Y = 80
Returning from ValueVar : G1 = 10; G2 = 80

Calling AllVars         : G1 = 10; G2 = 80
    Entering AllVars    :  X = 10;  Y = 80
    Leaving AllVars     :  X = 20;  Y = 320
Returning from AllVars  : G1 = 20; G2 = 320
```

Functions

Procedures are designed to do something for you. In contrast, functions are designed to *find* or *compute* something for you. Procedures may or may not report on what they do, or return a particular result. Functions, on the other hand, always return a result.

You may recall from algebra that a function is something that can be evaluated for particular starting values, or arguments. In simple cases, such an evaluation returns a single value. For example, the function

$y = x^2$

returns a value for any value of x — namely, the square of x. Thus,

$2.25 = 1.5^2$]

This result is said to be the value of the function at the starting value (1.5, in this case).

Similarly, suppose you have a function (**SQRT**) that computes the square root of values that you pass the function.

When you call **SQRT** with an argument, the function will do its work, and will return a single value — namely, the square root of the argument you passed. For example,

```
SQRT ( 81.0)
```

yields 9.0 as the value of the function for a starting value of 81.0. Similarly,

```
SQRT ( 6.25)
```

yields 2.5.

In Pascal, a *function* is a routine that always returns a specific value. Functions have the same components as procedures: a heading (or interface), a declaration section, and a body. The syntax for defining and for calling functions is different from the syntax for procedures, especially the syntax for the interface. Fortunately, the rules and principles involving parameters are the same for both types of routines.

Defining Functions in CodeWarrior Pascal

Let's look at some example functions — to illustrate the syntax for defining functions.

```
(* Return the larger of One, Two *)
FUNCTION Max ( One, Two : REAL) : REAL;
BEGIN
  IF One > Two THEN
    Max := One
  ELSE
    Max := Two;
END; (* FUNCTION Max : REAL *)

(* Return the smaller of One, Two *)
FUNCTION Min ( One, Two : REAL) : REAL;
BEGIN
  IF One > Two THEN
    Min := Two
  ELSE
    Min := One;
END; (* FUNCTION Min : REAL *)
```

The interface for a function is similar to a procedure heading, but differs in some important ways:

- The heading begins with the reserved word **FUNCTION**, rather than **PROCEDURE**.
- Following the name and the (optional) parameter list, the function heading ends with a return type — for example, **REAL**.

In CodeWarrior Pascal, a function can return any simple type, as well as strings and some types we haven't discussed yet. A function's return type is specified after a colon (:) in the heading.

A function body also differs from a procedure body in one very important way. Recall that a function returns its value. Syntactically, the function must get its value through an assignment statement somewhere in the function body.

Such an assignment sets the function equal to the value of some variable, value, or expression of the appropriate type. The function returns to the calling routine as soon as it is assigned a value. The function name can be on the left side of an assignment statement only within the function's body.

For example, in **Max**, the function would be finished, and would return to the calling routine as soon as either of the two assignment statements was executed.

TIP: A function heading must begin with the reserved word FUNCTION, and must end with a return type, specified following a colon. The function body must contain a statement in which a value of the return type is assigned to the function.

Calling a Function

As you've seen, procedure calls are very straightforward. To call a procedure, just specify its name and the appropriate argument list. Function calls work a bit differently. To understand the logic behind function calls, keep in mind that a function is simply a way of generating a specific value — namely, the value of the function. In fact, Pascal treats a function call as equivalent to a value.

For this reason, a function call can appear anywhere a value of its return type can appear. The following program illustrates several places where you might have a function call:

Listing 4.6

```
(* Program to illustrate function definitions and calls *)

PROGRAM FnCalls;

VAR Smaller, Larger, Val1, Val2 : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Return the larger of One, Two *)
  FUNCTION Max ( One, Two : REAL) : REAL;
  BEGIN
    IF One > Two THEN
      Max := One
    ELSE
      Max := Two;
  END; (* FUNCTION Max : REAL *)

   (* Return the smaller of One, Two *)
  FUNCTION Min ( One, Two : REAL) : REAL;
  BEGIN
    IF One > Two THEN
      Min := Two
    ELSE
      Min := One;
  END; (* FUNCTION Min : REAL *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)
```

```
BEGIN (* Main program *)
  GetReal ( 'Value 1?', Val1);
  GetReal ( 'Value 2?', Val2);
  (* right side of an assignment statement can have a REAL;
     therefore, a function call is allowed.
  *)
  Smaller := Min ( Val1, Val2);
  Larger := Max ( Val1, Val2);
  WRITELN ( 'Val1 = ', Val1 : 10 : 3,
            '; Val2 = ', Val2 : 10 : 3);
  WRITELN ( 'Smaller = ', Smaller : 10 : 3,
            '; Larger  = ', Larger : 10 : 3);
  (* You can write REAL values, so Min and Max can be
     arguments to another routine.
  *)
  WRITELN ( 'Min     = ', Min ( Val1, Val2) : 10 : 3,
            '; Max     = ', Max ( Val1, Val2) : 10 : 3);
  (* You can even put function calls into expressions
     in arguments.
  *)
  WRITELN ( 'Sum     = ',
            Min ( Val1, Val2)  + Max ( Val1, Val2) : 10 : 3);
  AwaitUser;
END.
```

The two places where you'll find function calls are on the right side of an assignment statement and as arguments to other procedures or functions.

When the program is executing and a function call is reached, the following happens:

- The calling routine is suspended temporarily, and control is switched to the function, which begins executing.
- The function executes in sequence, beginning at the top of the function body, and continues through the body until an appropriate value can be computed and assigned to the

function.

- After the function has been assigned a value, this value is returned to the calling routine, which resumes execution with the new value at the appropriate point in the program.

Because a function is nothing but a value, a function call can never appear as a statement by itself. That is, a function call cannot have the same format as a procedure call. To see why this is so, consider the fact that the following two statements would be equivalent if such a function call were allowed:

```
Min ( 39.0, 45.5);    (* a fancy way of saying 39 *)

39.0;                 (* a value is not a statement *)
```

Since a value by itself is not a statement, a function call (which is equivalent to a value) can't be a statement.

Parameters to Functions

Parameter passing works the same way for functions as it does for procedures. That is, you can pass by value or by reference. Because a function generally returns its main result directly, you'll have less need to change arguments. As a result, you're more likely to pass arguments by value to functions.

Predefined Procedures and Functions

Pascal includes about a dozen predefined routines that you can use in your programs; CodeWarrior Pascal adds several dozen more routines to this collection. The rest of this chapter contains brief descriptions of some of the simpler predefined routines. These routines will enable you to achieve a variety of results in programs until you've learned about control constructs and about more advanced data types.

Predefined Routines in Standard Pascal

This section contains descriptions of the routines that are predefined in standard Pascal. These routines are also available in CodeWarrior Pascal. Even when a routine has been predefined in both standard Pascal and in CodeWarrior Pascal, there may still be differences in what the routine can do. For example, many of the routines can take any whole number argument. In CodeWarrior Pascal, this means you can pass any of several different types: **INTEGER**, **LONGINT**, **UNSIGNEDWORD**, or **UNSIGNEDLONG**. In standard Pascal, such arguments can only be **INTEGER**s, since none of the other whole number types is predefined.

FUNCTION ABS

The **ABS** function returns the *absolute value* of its argument, which must be a signed whole number type or a **REAL**. The return type is the same type as the argument. Essentially, the absolute value of a number is its positive form.

For example, the following two statements produce the same result:

```
Result1 := ABS ( -39.0);   (* returns 39.0 *)
Result2 := ABS ( 39.0);    (* returns 39.0 *)
```

The following program lets you explore the **ABS** function:

Listing 4.7

```
(* Program to illustrate use of ABS function *)

PROGRAM AbsDemo;

VAR RVal, Result : REAL;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN  (* Main Program *)
  REPEAT
    GetReal ( 'Value (> 500 to stop)? ', RVal);
    Result := ABS ( RVal);
    WRITELN ( 'ABS ( ', RVal : 8:3, ') = ', Result : 8 : 3);
  UNTIL RVal > 500.0;
  AwaitUser;
END.
```

FUNCTION ARCTAN

The **ARCTAN** function returns a trigonometric value — namely, the angle that corresponds to a particular *tangent* value. Both the argument and the return type are **REAL**s.

The following program lets you explore this function:

Listing 4.8

```
(* Program to illustrate use of ARCTAN function *)

PROGRAM ArctanDemo;

VAR RVal, Result : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN  (* Main Program *)
  REPEAT
    GetReal ( 'Value (> 10.0 to stop)? ', RVal);
    Result := ArcTan ( RVal);
    WRITELN ( 'ArcTan ( ', RVal : 8:3, ') = ', Result : 8 : 3);
  UNTIL RVal > 10.0;
  AwaitUser;
END.
```

FUNCTION CHR

The **CHR** function returns the character having the specified ordinal value. The ordinal value represents an ASCII code in CodeWarrior Pascal. For example, the following program writes several things — all on the same line.

Listing 4.9

```
(* Program to illustrate use of CHR function *)

PROGRAM CHRDemo;

CONST CarriageReturn = 13;
      BackSpace = 8;
VAR Result : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  Result := 100;
  WRITE ( Result : 30);
   (* send a backspace, to move back one column,
      then write a new digit at the end of the number
      — which is where the cursor is after the Backspace.
  *)
  WRITE ( CHR ( BackSpace), 1);
   (* send a carriage return but no linefeed,
      so cursor moves to start of same line.
  *)
  WRITE ( CHR ( CarriageReturn));
  WRITE ( Result : 20);
  WRITE ( CHR ( BackSpace), 2);
  WRITE ( CHR ( CarriageReturn));
  WRITE ( Result : 10);
  WRITE ( CHR ( BackSpace), 3);
  WRITE ( CHR ( CarriageReturn));
  AwaitUser;
END.
```

This program does the following:

Writes **100** using a large field width, but without moving the cursor to the next line. The field width is smaller each time this action is carried out.

- Writes a BackSpace character (ASCII 8), which moves the cursor back over the last digit in the **100**.
- Writes a new digit to replace the last digit. This digit changes each time the action is carried out.
- Writes a Carriage Return character (ASCII 13), which moves the cursor to the leftmost column of the current line. (A Line Feed would be needed to move to the next line.)

The middle two actions are performed with a single call to **WRITE**. This sequence of actions is repeated three times. Each time, the value (100) is written in a smaller field width, and each time the last digit in the number is replaced by a different digit.

See **ORD** for a related function.

FUNCTION COS

The **COS** function returns the cosine of the angle specified in the (**REAL**) argument. This will be a **REAL** value between -1.0 and 1.0, inclusive. The angle is specified using radians, rather than degrees. Radian measures express angles as fractions or multiples of π (approximately 3.14159265). For example, 180 degrees is equivalent to π radians; 1 radian is about 57 degrees.

The following program lets you explore the **COS** function. Run the program to see how this trigonometric function behaves.

Listing 4.10

```
(* Program to illustrate use of COS function *)

PROGRAM CosDemo;

VAR Cosine, Angle : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN
  REPEAT
    GetReal ( 'Angle (in radians; > 100.0 to stop)?', Angle);
    Cosine := COS ( Angle);
    WRITELN ( 'Cosine of ', Angle : 6 : 4,
              ' radians = ', Cosine : 6 : 4);
  UNTIL Angle > 100.0;
  AwaitUser;
END.
```

See **SIN** for another trigonometric function.

FUNCTION EXP

Function **EXP** returns the result of raising e (approximately 2.7181828]) to the specified power. For example, the following program computes **EXP (Val)** for values the user enters.

Listing 4.11

```
(* Program to illustrate use of EXP function *)

PROGRAM ExpDemo;

VAR Val, Result : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN
  REPEAT
    GetReal ( 'Value? (> 10 to stop) ', Val);
    Result := EXP ( Val);
    WRITELN ( 'EXP (', Val : 10 : 3, ') = ', Result : 10 : 3);
  UNTIL Val > 10.0;
  AwaitUser;
END.
```

FUNCTION LN

Function **LN** returns the *natural logarithm* (logarithm to base *e*) of **Val**, which must be greater than 0.0. The natural logarithm of a value is the power to which *e* must be raised in order to get the number.

For example, e^2 (approximately 7.389056). This means that **LN** (7.389056) is 2, since *e* must be raised to this power to yield 7.389056.

The following program extends the program for **EXP** to include the **LN** function as well. The results indicate that **EXP** and **LN** are inverse functions.

Listing 4.12

```
(* Program to illustrate use of EXP and LN functions *)

PROGRAM ExpAndLnDemo;

VAR Val, Result, LResult : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)
```

```
BEGIN
  REPEAT
    GetReal ( 'Value? (< 0 to quit)', Val);
    Result := Exp ( Val);
    WRITELN ( 'EXP (', Val : 10 : 3, ') = ', Result : 10 : 3);
    LResult := LN ( Result);
    WRITELN ( 'LN  (', Result : 10 : 3, ') = ', LResult:10:3);
  UNTIL Val < 0.0;
  AwaitUser;
END.
```

FUNCTION ODD

Function **ODD** returns **TRUE** if **Val** is an odd number — that is, leaves a remainder of 1 when divided by 2. The argument is a whole number type.

The following program illustrates how to use **ODD**. The program generates random **INTEGER** values. If the value is odd, the program writes a period; if the value is even, the program writes a space.

Run the program several times. Try setting different large values (for example, values over 5000) for the maximum value to generate (**MaxVal**. You'll find that the proportion of odd values is generally pretty close to 50%. That is, about half of the random integer values generated will be odd, and the other half will be even.

Then run the program using different small values (for example, less than 20). Try some cases in which **MaxVal** is odd, and others in which **MaxVal** is even. You'll notice that the proportion of odd values fluctuates much more when you have a smaller range of values.

Listing 4.13

```
(* Program to illustrate use of ODD function *)
```

```
PROGRAM OddDemo;
USES Windows;

CONST MaxTrials = 1840;
VAR Count, OddCount, MaxVal, Result, RandSeed : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN
   (* Get and set seed value for random number generator *)
  GetInt ( 'Integer seed for random number generator', RandSeed);
  SRAND ( RandSeed);
  OddCount := 0;
  GetInt ( 'Largest value to generate? (< 0 for MAXINT)',
               MaxVal);
  IF MaxVal < 0 THEN
    MaxVal := MAXINT;
  FOR Count := 1 TO MaxTrials DO
  BEGIN
    Result := RAND MOD MaxVal;
     (* Note that function can be used
        wherever a BOOLEAN is valid.
    *)
    IF ODD ( Result) THEN
    BEGIN
      WRITE ( '.');
       (* Increment the odd value counter by 1 *)
      INC ( OddCount);
    END
    ELSE
      WRITE ( ' ');
  END;
  (* determine and display the proportion of odd values *)
  WRITE ( OddCount, ' odd values out of ', MaxTrials);
  WRITELN ( '; (', (OddCount / MaxTrials) * 100 :5:2, '%)');
  AwaitUser;
END.
```

FUNCTION ORD

Function **ORD** returns the ordinal value of the argument, which can be of any ordinal type. For **CHAR**s, this value is the ASCII code; for whole number types, the result is simply the value; for **BOOLEAN**s, the result is 0 or 1 for **FALSE** or **TRUE**, respectively; for enumerated types, the result is one less than the value's position in the list of values. That is, for enumerated types, the first element has value 0, the second has value 1, and so forth.

The following program illustrates the use of the **ORD** function.

Listing 4.14

```
(* Program to illustrate use of ORD function *)

PROGRAM OrdTest;

TYPE MyType = ( vala, valb, valc);
VAR I1 : INTEGER;
    C1 : CHAR;
    B1 : BOOLEAN;
    M1 : MyType;
```

```
  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  I1 := -1;
  C1 := 'a';
  B1 := FALSE;
  (* for enumerated types, first element has ordinal value 0,
     second element has ordinal value 1, etc.
  *)
  M1 := valc;
  WRITELN ( 'ORD ( I1) = ', ORD ( I1));  (* returns -1 *)
  WRITELN ( 'ORD ( C1) = ', ORD ( C1));  (* returns 97 *)
  WRITELN ( 'ORD ( B1) = ', ORD ( B1));  (* returns 0 *)
  WRITELN ( 'ORD ( M1) = ', ORD ( M1));  (* returns 2 *)
  AwaitUser;
END.
```

See **CHR** for a related function.

FUNCTION PRED

Function **PRED** returns the value preceding the one passed as an argument. The argument can be of any ordinal type. The result is undefined if you ask for the predecessor of the minimum value.

The following program illustrates the use of the predecessor function.

Listing 4.15

```
(* Program to illustrate use of PRED function *)

PROGRAM PredDemo;

TYPE MyType = ( vala, valb, valc);
VAR I1 : INTEGER;
    C1 : CHAR;
    B1 : BOOLEAN;
    M1 : MyType;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  I1 := -32768;
  C1 := 'q';
  B1 := FALSE;
  (* for user defined types, first element has ordinal value 0,
     second element has ordinal value 1, etc.
  *)
  M1 := valc;
  WRITELN ( 'PRED ( I1) = ', PRED ( I1));
  WRITELN ( 'PRED ( C1) = ', PRED ( C1));
  WRITELN ( 'PRED ( B1) = ', PRED ( B1));
  (* You can't write the value of an enumerated type;
     one way to determine whether the PRED did anything
     is to write the new ordinal value.
     Since ORD ( valc) = 2, the new value should be 1.
  *)
  WRITELN ( 'PRED ( M1) = ', ORD ( PRED ( M1)));
  AwaitUser;
END.
```

See **SUCC** for a related function.

PROCEDUREs READ and READLN

READ and **READLN** are the procedures for doing input from files or from the standard input (keyboard). You've already learned about these two routines in Chapter 3.

FUNCTION ROUND

Function **ROUND** converts a **REAL** to a **LONGINT** by rounding the number's fractional part. The function rounds downward if the fractional part is less than 0.5, and rounds upward otherwise. In CodeWarrior Pascal, the result is returned as a **LONGINT**; in standard Pascal, the result is an **INTEGER**.

For example, the first statement in the following listing assigns the value 39 to **Result**, the second assigns the value 40.

```
Result := ROUND ( 39.46);   (* assigns 39 *)

Result := ROUND ( 39.86);   (* assigns 40 *)
```

This function is important because it lets you get a value of one type (**LONGINT**) from a value of a different type (**REAL**).

You can use the following program to explore the **ROUND** function. Try entering several both positive and negative values. What happens to a value such as 3.5? With that information, try to predict what will happen to -3.5?

Listing 4.16

```
(* Program to illustrate use of ROUND function *)

PROGRAM RoundDemo;

VAR RVal, Result : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN  (* Main Program *)
  REPEAT
    GetReal ( 'Value (> 500 to stop)? ', RVal);
    Result := ROUND ( RVal);
    WRITELN ( 'ROUND ( ', RVal : 8:3, ') = ', Result : 8 : 3);
  UNTIL RVal > 500.0;
  AwaitUser;
END.
```

FUNCTION SIN

Function **SIN** returns the sine of the angle specified in the (**REAL**) argument. This will be a **REAL** value between -1.0 and 1.0, inclusive. The angle is specified using radians, rather than degrees. Radian measures express angles as fractions or multiples of π (approximately 3.14159265). For example, 180 degrees is equivalent to π radians; 1 radian is about 57 degrees.

The following program lets you explore the **SIN** function. Run the program for various radian values between 0 and 2π (approximate;ely 6.2831853). Can you determine the behavior of the sine function within this range of values? See the discussion of the **SQR** function for another program involving **SIN**.

Listing 4.17

```
(* Program to illustrate use of SIN function *)

PROGRAM SinDemo;

VAR Sine, Angle : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN
  REPEAT
    GetReal ( 'Angle (in radians; > 100 to stop)?', Angle);
    Sine := SIN ( Angle);
    WRITELN ( 'Sine of ', Angle : 6 : 4,
              ' radians = ', Sine : 6 : 4);
  UNTIL Angle > 100;
  AwaitUser;
END.
```

See **COS** for another trigonometric function.

FUNCTION SQR

Function **SQR** returns the square of the argument — that is **Val** * **Val**. The returned value is of the same type as the argument, which must be a whole number type or a **REAL**. Thus, if you pass in an **INTEGER** argument, the returned value will also be an **INTEGER**. With a function such as **SQR**, you'll need to be careful of overflow when using whole numbers.

The following program lets you explore the **SQR** function. The program uses **LONGINT** values. What's the largest value you can enter without overflow?

Listing 4.18

```
(* Program to illustrate use of SQR function *)

PROGRAM SqrDemo;

VAR LVal, Result : LONGINT;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)
  PROCEDURE GetLongInt ( Message : STRING;
                         VAR Value : LONGINT);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetLongInt *)

BEGIN  (* Main Program *)
  REPEAT
    GetLongInt ( 'Value (< -500 to stop)? ', LVal);
    Result := SQR ( LVal);
    WRITELN ( 'SQR ( ', LVal:10, ') = ', Result : 15);
  UNTIL LVal < -500.0;
  AwaitUser;
END.
```

The following program uses the **SQR** function to help illustrate a relationship between the sine and cosine values for a particular value.

Listing 4.19

```
(* Program to illustrate use of SQR function,
   and a relationship between the SIN and COS functions.
*)

PROGRAM SqrAndTrigDemo;

VAR Sine, Cosine, Angle : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN
  WRITELN ( '  SIN(x)  ', '  COS(X)  ',
            'SQR(Sine) + SQR(Cosine)');
  WRITELN;
  REPEAT
    GetReal ( 'Angle (> 100 to stop)?', Angle);
    Sine := SIN ( Angle);
    Cosine := COS ( Angle);
    WRITELN ( Sine : 10 : 4, Cosine : 10 : 4,
              SQR ( Sine) + SQR ( Cosine) : 17 : 4);
  UNTIL Angle > 100;
  AwaitUser;
END.
```

FUNCTION SQRT

Function **SQRT** returns the square root of the argument, which must be a non-negative value. The argument can be any whole number type or a **REAL**. The returned value is always a **REAL**.

The following program lets you explore the **SQRT** function:

Listing 4.20

```
(* Program to illustrate use of SQRT function *)

PROGRAM SqrtDemo;

VAR RVal, Result : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN  (* Main Program *)
  GetReal ( 'Value (< 0 to stop)? ', RVal);
  WHILE RVal > 0 DO
  BEGIN
    Result := SQRT ( RVal);
    WRITELN ( 'SQRT ( ', RVal : 8:3, ') = ', Result : 8 : 3);
    GetReal ( 'Value (< 0 to stop)? ', RVal);
  END; (* WHILE RVal > 0 *)
  AwaitUser;
END.
```

FUNCTION SUCC

Function **SUCC** returns the value following the one passed as an argument. The argument can be of any ordinal type. The result is undefined if you ask for the successor of the maximum value.

The following program illustrates the use of the successor function.

Listing 4.21

```
(* Program to illustrate use of SUCC function *)

PROGRAM SuccDemo;

TYPE MyType = ( vala, valb, valc);
VAR I1 : INTEGER;
    C1 : CHAR;
    B1 : BOOLEAN;
    M1 : MyType;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  I1 := 32767;
  C1 := 'q';
  B1 := TRUE;
  (* for user defined types, first element has ordinal value 0,
     second element has ordinal value 1, etc.
  *)
  M1 := valc;
  WRITELN ( 'SUCC ( I1) = ', SUCC ( I1));
  WRITELN ( 'SUCC ( C1) = ', SUCC ( C1));
  WRITELN ( 'SUCC ( B1) = ', SUCC ( B1));
  (* You can't write the value of an enumerated type;
     one way to determine whether the SUCC did anything
     is to write the new ordinal value.
     Since ORD ( valc) = 2, the new value should be 3,
     although there is no enumerated value with this
     ordinal value.
  *)
  WRITELN ( 'SUCC ( M1) = ', ORD ( SUCC ( M1)));
  AwaitUser;
END.
```

See **PRED** for a related function.

FUNCTION TRUNC

Function **TRUNC** converts a **REAL** to a **LONGINT** by dropping the number's fractional part. The result is a **LONGINT** in CodeWarrior Pascal, and an **INTEGER** in standard Pascal. This function always rounds downward. Thus, both of the assignments in the following listing would assign 39 to **Result**.

```
Result := TRUNC ( 39.46);   (* assigns 39 *)

Result := TRUNC ( 39.86);   (* assigns 39 *)
```

Like **ROUND**, this function enables you to convert a value from **REAL** to a whole number type. The following program lets you compare the behavior of **TRUNC** and **ROUND**. How do these two functions differ for values such as 3.5? 3.8? -3.8?

Listing 4.22

```
(* Program to illustrate use of TRUNC function *)

PROGRAM TruncDemo;

VAR RVal, Result : REAL;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN  (* Main Program *)
  REPEAT
    GetReal ( 'Value (> 500 to stop)? ', RVal);
    Result := TRUNC ( RVal);
    WRITELN ( 'TRUNC ( ', RVal : 8:3, ') = ', Result : 8 : 3);
    Result := ROUND ( RVal);
    WRITELN ( 'ROUND ( ', RVal : 8:3, ') = ', Result : 8 : 3);
  UNTIL RVal > 500.0;
  AwaitUser;
END.
```

WRITE and *WRITELN* Procedures

WRITE and **WRITELN** are the routines for writing material to the screen or to a file. You've already learned about these routines in earlier chapters.

Routines Predefined in CodeWarrior Pascal

The remaining routines in this chapter are predefined in CodeWarrior Pascal, but not in standard Pascal. Many of these routines involve strings.

FUNCTION CONCAT

Function **CONCAT** creates a larger string by combining individual string arguments. The larger string is returned. When concatenating two strings (say **Str1** and **Str2**), the result is a longer string which contains both strings, with **Str1** first. This function can take two or more string arguments.

The following program illustrates the use of the **CONCAT** function.

Listing 4.23

```
(* Program to illustrate use of CONCAT function *)

PROGRAM ConcatDemo;
USES Windows;

VAR Str1, Str2, Str3, Str4, LargeStr : STRING;
    Result, RandSeed : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN
  (* Get and set seed value for random number generator *)
  GetInt ( 'Integer seed for random number generator', RandSeed);
  SRAND ( RandSeed);
  Result := RAND;
  GetString ( 'Str1?', Str1);
  GetString ( 'Str2?', Str2);
  GetString ( 'Str3?', Str3);
  GetString ( 'Str4?', Str4);
  IF ODD ( Result) THEN
  BEGIN
    WRITELN ( 'Concatenating Str1 through Str4');
    LargeStr:= CONCAT ( Str1, Str2, Str3, Str4);
  END
  ELSE
  BEGIN
    WRITELN ( 'Concatenating Str4 through Str1');
    LargeStr := CONCAT ( Str4, Str3, Str2, Str1);
  END;
  WRITELN ( LargeStr);
  AwaitUser;
END.
```

The program gets strings from the user and, after reading the strings, it gets a random value. If this value is odd, the program concatenates the strings in order; if the value is even, the program concatenates them in reverse order. In the latter case, the resulting string consists of **Str4**, Str3, **Str2**, and **Str1**, in that order. Note that the order of the strings is reversed in the latter case, but not the order of the characters within the strings.

FUNCTION COPY

The **COPY** function returns a substring copied from **Str**. The copied substring begins with character **Start** of **Str**. At most **NrChars** characters are copied from **Str**. For example, the following call would copy the first seven characters of **Str** to **ResultStr**:

```
ResultStr := COPY ( Str, 1, 7);
```

If the end of **Str** is reached before enough characters have been copied, then **COPY** returns a shorter substring. If **Start** is larger than the length of **Str**, the function returns an empty string.

Run the following program to copy various substrings from strings that you specify. Try to copy more characters than you have in your string. For example, if **SVal** is 10 characters, tell the program you want to copy 12 characters. Similarly, try to start copying from a point beyond the end of the string. For example, if **SVal** is 10 characters, try to copy from position 11.

Listing 4.24

```
(* Program to illustrate use of COPY function *)

PROGRAM CopyDemo;
```

```
CONST EmptyString = '';
VAR SVal, Result : STRING;
    StartIndex, NrCh : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

BEGIN  (* Main Program *)
  GetString ( 'String (ENTER to stop)? ', SVal);
  WHILE SVal <> EmptyString DO
  BEGIN
    GetInteger ( 'Starting position? ', StartIndex);
    GetInteger ( '# chars to copy? ', NrCh);
    Result := COPY ( SVal, StartIndex, NrCh);
    WRITELN ( 'Substring ', StartIndex : 2, ' .. ',
              NrCh : 2, ' = ', Result);
    GetString ( 'String (ENTER to stop)? ', SVal);
  END;  (* WHILE SVal <> EmptyString *)
  AwaitUser;
END.
```

PROCEDURE DEC

The **DEC** procedure decrements (reduces) its argument by one. The parameter must be an integral type. For such arguments, the procedure is equivalent to the **PRED** function, which returns the previous value or the ordinal type specified. The major difference between the two routines is that **DEC** does not take **CHAR** or **BOOLEAN** arguments.

The following program illustrates the **DEC** procedure. Note the use of the **PRED** function for dealing with the **CHAR** and **BOOLEAN** variables.

Listing 4.25

```
(* Program to illustrate use of DEC procedure *)

PROGRAM DecDemo;

VAR I1 : INTEGER;
    L1 : LONGINT;
    C1 : CHAR;
    B1 : BOOLEAN;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Display one value of each of four types *)
  PROCEDURE DispTypes ( IVal : INTEGER;
                        LVal : LONGINT;
                        CVal : CHAR;
                        BVal : BOOLEAN);
  BEGIN
```

```
    WRITELN ( 'Integer value = ', IVal);
    WRITELN ( 'LongInt value = ', LVal);
    WRITELN ( 'Char value    = ', CVal);
    WRITELN ( 'Boolean value = ', BVal);
  END;  (* DispTypes *)

BEGIN
  (* Initialize values *)
  (* You can have more than one statement per line *)
  I1 := 30;              L1 := 55000;
  C1 := 'g';             B1 := TRUE;
  (* Decrement each value by 1 *)
  DEC ( I1);             DEC ( L1);
  C1 := PRED ( C1);      B1 := PRED( B1);
  DispTypes ( I1, L1, C1, B1);
  AwaitUser;
END.
```

As you can see from this listing, you can have more that one statement on a single line. Lines with multiple statements are more difficult to read, so we'll avoid such overworked lines in other programs.

The following listing contains the output from the example program:

```
Integer value = 29
LongInt value = 54999
Char value    = f
Boolean value = FALSE
```

PROCEDURE DELETE

CodeWarrior Pascal's **DELETE** procedure removes a substring from **Str**. The substring begins at character **StartingIndex** and continues for **NrToDelete** characters altogether.

If there are not enough characters in the string to delete, then the procedure deletes from character **StartingIndex** to the end of the string. If **StartingIndex** is greater than the length of **Str** then nothing is deleted.

For example, the following statement would delete the first 9 characters from **StrToShorten**:

```
Delete ( StrToShorten, 1, 9);
```

The following program lets you explore the **DELETE** procedure. Try deleting past the end of your **SVal** string. One way to do this is to start deleting at a position past the end of the string; the other way is to start somewhere within **SVal** and then delete more characters than **SVal** between the starting position and the end of the string.

Listing 4.26

```
(* Program to illustrate use of DELETE function *)

PROGRAM DeleteDemo;

CONST EmptyString = '';
VAR SVal : STRING;
    StartIndex, NrCh : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

BEGIN  (* Main Program *)
  GetString ( 'String (ENTER to end loop)? ', SVal);
  WHILE SVal <> EmptyString DO
  BEGIN
    GetInteger ( 'Starting position? ', StartIndex);
    GetInteger ( '# chars to delete? ', NrCh);
    DELETE ( SVal, StartIndex, NrCh);
    WRITELN ( 'Remaining string = ', SVal);
    GetString ( 'String (ENTER to end loop)? ', SVal);
    WRITELN ( 'dbTMP: New SVal = ', SVal);
  END;  (* WHILE SVal <> EmptyString *)
  AwaitUser;
END.
```

PROCEDURE INC

The **INC** procedure increments (increases) its argument by one. The parameter must be an integer type. For such arguments, the procedure is equivalent to the **SUCC** function, which returns the next value for the ordinal type specified.

The following program illustrates the **INC** procedure. (This is the same program as for **DEC**, except that all references to **DEC** have been changed to **INC**.) Note again, that **SUCC** is used to handle the **CHAR** and **BOOLEAN** variables.

Listing 4.27

```
(* Program to illustrate use of INC procedure *)

PROGRAM IncDemo;

VAR I1 : INTEGER;
    L1 : LONGINT;
    C1 : CHAR;
    B1 : BOOLEAN;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Display one value of each of four types *)
  PROCEDURE DispTypes ( IVal : INTEGER;
                        LVal : LONGINT;
                        CVal : CHAR;
                        BVal : BOOLEAN);
  BEGIN
    WRITELN ( 'Integer value = ', IVal);
    WRITELN ( 'LongInt value = ', LVal);
    WRITELN ( 'Char value    = ', CVal);
    WRITELN ( 'Boolean value = ', BVal);
  END;  (* DispTypes *)

BEGIN
```

```
  (* Initialize values *)
  (* You can have more than one statement per line *)
  I1 := 30;              L1 := 55000;
  C1 := 'g';             B1 := TRUE;
  (* Increment each value by 1 *)
  INC ( I1);             INC ( L1);
  C1 := SUCC ( C1);      B1 := SUCC ( B1);
  DispTypes ( I1, L1, C1, B1);
  AwaitUser;
END.
```

The following listing contains the output from the program. Compare this output with that from program **DecDemo**, which illustrates the use of the **DEC** procedure.

```
Integer value = 31
LongInt value = 55001
Char value    = h
Boolean value = TRUE
```

PROCEDURE INSERT

The **INSERT** procedure inserts **StrToAdd** into **WhereToAdd**, beginning at position **StartingPos**. If the resulting string would be longer than 255 characters, only the first 255 characters are saved. For example, if **MyString** has the value `there, duckies' then the statement

```
(* Assume MyString = 'there, duckies' *)
INSERT ( 'Hello ', MyString, 1);
```

makes **MyString** equal to `Hello there, duckies' after the call.

You can use the following program to explore the **INSERT** procedure:

Listing 4.28

```
(* Program to illustrate use of INSERT procedure *)

PROGRAM InsertDemo;

CONST EmptyString = '';
VAR SVal, ToAdd : STRING;
    StartIndex : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

BEGIN  (* Main Program *)
  GetString ( 'String (ENTER to end loop)? ', SVal);
  WHILE SVal <> EmptyString DO
```

```
                    BEGIN
                      GetString ( 'String to add? ', ToAdd);
                      GetInteger ( 'Starting Position? ', StartIndex);
                      INSERT ( ToAdd, SVal, StartIndex);
                      WRITELN ( 'Result = ', SVal);
                      GetString ( 'String (ENTER to end loop)? ', SVal);
                    END;  (* WHILE SVal <> EmptyString *)
                    AwaitUser;
END.
```

FUNCTION LENGTH

Function **LENGTH** returns the number of characters currently contained in string **Str**. For example, if **MyStr** is `this long' then

```
Result := LENGTH ( MyStr);
```

assigns 9 to **Result**.

The following program lets you explore the **LENGTH** function:

Listing 4.29

```
(* Program to illustrate use of LENGTH function *)

PROGRAM LengthDemo;
```

```
CONST EmptyString = '';
VAR SVal : STRING;
    Result : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

BEGIN  (* Main Program *)
  GetString ( 'String (ENTER to stop)? ', SVal);
  WHILE SVal <> EmptyString DO
  BEGIN
    Result := LENGTH ( SVal);
    WRITELN ( SVal, ' is ', Result:2, ' characters long');
    GetString ( 'String (ENTER to stop)? ', SVal);
  END;  (* WHILE SVal <> EmptyString *)
  AwaitUser;
END.
```

FUNCTION POS

Function **POS** Searches for **StrToFind** in **StrToSearch**. If the substring is found in **StrToSearch**, the function returns the starting position of the substring's *first* occurrence. If the substring is not found in **StrToSearch**, the function returns 0.

For example, if **Pattern** is `for' and **Source** is `California' then the following statement

```
(* Assume Pattern = 'for' and Source = 'California' *)
Result := POS ( Pattern, Source);
```

assigns 5 to **Result**.

You can use the following program to explore the **POS** function:

Listing 4.30

```
(* Program to illustrate use of POS function *)

PROGRAM PosDemo;

CONST EmptyString = '';
VAR SVal, ToFind : STRING;
    Result : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
```

```
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

BEGIN  (* Main Program *)
  GetString ( 'String (ENTER to stop)? ', SVal);
  WHILE SVal <> EmptyString DO
  BEGIN
    GetString ( 'String to find? ', ToFind);
    Result := POS ( ToFind, SVal);
    WRITELN ( 'Starting position = ', Result);
    GetString ( 'String (ENTER to stop)? ', SVal);
  END;  (* WHILE SVal <> EmptyString *)
  AwaitUser;
END.
```

FUNCTION RAND

Function **RAND** generates and returns a pseudorandom integer between 0 and **RAND_MAX**, which is defined as equal to **MAXINT**, or 32767, for CodeWarrior Pascal.

The random number generator being used must be initialized, or seeded, and the actual values generated in a sequence depend on the initialization seed. That is, **RAND** always generates the same sequence of values when starting from the same seed. This is one of the reasons why the returned value is considered pseudorandom.

To use **RAND** in a program, you must include a **USES Windows;** statement at the start of your program — ie, immediately after the **PROGRAM ....** line.

See also **SRAND**.

FUNCTION SIZEOF

Function **SIZEOF** returns the amount of storage allocated to the variable or type specified. The result represents the number of bytes allocated for the argument. The argument can be a type specifier or a variable name.

The following program displays the storage allocated for the types you've seen do far:

Listing 4.31

```
PROGRAM SizeDemo;

VAR I1 : INTEGER;
    L1 : LONGINT;
    UW1 : UNSIGNEDWORD;
    UL1 : UNSIGNEDLONG;
    C1 : CHAR;
    Bo1 : BOOLEAN;
    Stg : STRING;
    R1 : REAL;
    LR1 : LONGREAL;
    S1 : SINGLE;         (* same as a REAL *)
    D1 : DOUBLE;         (* same as a LONGREAL *)

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  WRITELN ( 'INTEGER         : ', SIZEOF ( INTEGER) : 3, '
bytes; ',
            '   I1 : ', SIZEOF ( I1) : 3, ' bytes');
  WRITELN ( 'LONGINT         : ', SIZEOF ( LONGINT) : 3, '
bytes; ',
            '   L1 : ', SIZEOF ( L1) : 3, ' bytes');
  WRITELN ( 'UNSIGNEDWORD    : ', SIZEOF ( UNSIGNEDWORD) : 3, '
bytes; ',
            '   UW1: ', SIZEOF ( UW1) : 3, ' bytes');
  WRITELN ( 'UNSIGNEDLONG     : ', SIZEOF ( UNSIGNEDLONG) : 3, '
bytes; ',
            '   UL1: ', SIZEOF ( UL1) : 3, ' bytes');
  WRITELN ( 'CHAR             : ', SIZEOF ( CHAR) : 3, ' bytes; ',
            '   C1 : ', SIZEOF ( C1) : 3, ' bytes');
  WRITELN ( 'BOOLEAN          : ', SIZEOF ( BOOLEAN) : 3, '
bytes; ',
            '   Bo1: ', SIZEOF ( Bo1) : 3, ' bytes');
  WRITELN ( 'STRING           : ', SIZEOF ( STRING) : 3, ' bytes;
',
            '   Stg: ', SIZEOF ( Stg) : 3, ' bytes');
  WRITELN ( 'REAL             : ', SIZEOF ( REAL) : 3, ' bytes; ',
            '   R1 : ', SIZEOF ( R1) : 3, ' bytes');
  WRITELN ( 'LONGREAL         : ', SIZEOF ( LONGREAL) : 3, '
bytes; ',
            '   LR1: ', SIZEOF ( LR1) : 3, ' bytes');
  WRITELN ( 'SINGLE           : ', SIZEOF ( SINGLE) : 3, ' bytes;
',
            '   S1 : ', SIZEOF ( S1) : 3, ' bytes');
  WRITELN ( 'DOUBLE           : ', SIZEOF ( DOUBLE) : 3, ' bytes;
',
            '   D1 : ', SIZEOF ( D1) : 3, ' bytes');
  AwaitUser;
END.
```

Notice that the program produces the same result, regardless of whether you ask about a type or about a variable of that type. In Chapter 8, You'll see why a **STRING** is allocated 256 bytes but can only have 255 characters.

This program produces the following output.

```
INTEGER :   2 bytes;     I1 :   2 bytes
BYTE    :   1 bytes;     By1:   1 bytes
SHORTINT:   1 bytes;     S1 :   1 bytes
LONGINT :   4 bytes;     L1 :   4 bytes
```

```
WORD    :   2 bytes;    W1 :   2 bytes
CHAR    :   1 bytes;    C1 :   1 bytes
BOOLEAN :   1 bytes;    Bo1:   1 bytes
STRING  : 256 bytes;    Str: 256 bytes
REAL    :   6 bytes;    R1 :   6 bytes
```

PROCEDURE SRAND

Function **SRAND** initializes, or seeds, the random number generator with an integer value. This initialized generator is used by **RAND** to generate a pseudorandom **INTEGER**.

The values generated depend on the seed. The random number generator will always generate the same sequence when starting from a particular seed.

Before calling **RAND**, you should always call **SRAND**.

FUNCTION UPCASE

Function **UPCASE** returns the uppercase version of the character passed as an argument. Note that the argument itself is passed by value, so that the argument is not changed in the calling routine.

The following program lets you explore the **UPCASE** function:

Listing 4.32

```
(* Program to illustrate use of UPCASE function *)

PROGRAM UpcaseDemo;

CONST EmptyString = '';
VAR SVal : STRING;
    Index, Lgth : INTEGER;
```

```
  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

BEGIN  (* Main Program *)
  GetString ( 'String (ENTER to stop)? ', SVal);
  WHILE SVal <> EmptyString DO
  BEGIN
    WRITE ( SVal, ' : ');
    Lgth := LENGTH ( SVal);
    FOR Index := 1 TO Lgth DO
      SVal [ Index] := UPCASE ( SVal [ Index]);
    WRITELN ( SVal);
    GetString ( 'String (ENTER to stop)? ', SVal);
  END;  (* WHILE SVal <> EmptyString *)
  AwaitUser;
END.
```

Summary

In this chapter, you learned about procedures and functions. In particular, you learned how to define and call routines; you also learned how to pass parameters by value and by reference. As you create more example programs, you'll become more comfor with each of these methods.

The chapter also contained a whirlwind tour of some of CodeWarrior Pascal's predefined routines, which are available for use in your programs. You'll see more of these functions, and you 'll learn about other predefined routines as we cover more of Pascal.

In the next chapter, you'll learn about the structures Pascal provides for controlling the sequence in which your program executes.

5

CodeWarrior Pascal's Control Structures

In this chapter, you'll learn about the structures Pascal provides for controlling what your routines and programs do at various points in their execution. You'll learn about Pascal's constructs for selection (taking conditional actions) and iteration (repeating, or looping, actions). (An *iteration* is a trial or a repetition.)

Pascal provides two types of statements for making selections during execution, and three types of iteration statements. In a *selection construct*, a particular action may be taken *if* a condition holds; otherwise, either a different action or no action may be taken. In an *iteration construct*, an action may be repeated (possibly with variation). This repetition may continue for a fixed number of times, or it may continue *while* a condition holds or *until* some specified condition becomes true.

We'll look at one of the iteration structures first. This will enable you to work with example programs that do more interesting things. Following this, you'll learn about Pascal's selection constructs, and then its other iteration constructs.

Simple Repetition: The FOR Statement

In Chapter 4, you created procedure **CountVals** to count to a specified value. To accomplish its task, the procedure uses a **FOR** loop, which repeats as often as is specified in the parameter. This loop is one of the three iteration constructs Pascal provides.

In Pascal, the **FOR** loop is used when you know how often something needs to be done. For example, the following program repeats the same sequence of actions **MaxVals** times. This repetition is controlled by a **FOR** loop. In addition, the procedures defined in the program have their own **FOR** loops, which control various actions taken by these routines.

Listing 5.1

```
(* Program to illustrate use of FOR loops *)

PROGRAM ForDemo1;
USES Windows;

CONST MaxVals = 10;
      MaxItems = 5;
VAR Index, Result, RandSeed :INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Generate and display values based on random integers *)
  PROCEDURE DispVariants ( Value : INTEGER);
  CONST MaxVariants = 15;
  VAR Count : INTEGER;
  BEGIN
    (* Value is the value generated by the main program *)
    WRITE ( Value : 5);
    (* Write MaxVariants values *)
    FOR Count := 11 TO MaxVariants DO
      WRITE ( Value + ( RAND MOD (Count * 100)) : 5);
    WRITELN;
  END; (* DispVariants *)

  PROCEDURE PutHeadings;
  VAR Count : INTEGER;
  BEGIN
    WRITE ( ' ' : 5, ' Val ');
    FOR Count := 1 TO MaxItems DO
      WRITE ( Count * 100 + 1000 : 5);
    WRITELN;
    (* write 60 dashes, to make a line *)
    FOR Count := 1 TO MaxItems * 7 DO
      WRITE ( '-');
    WRITELN;
  END; (* PutHeadings *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN (* Main *)
  GetInt ( 'Integer for random number generator', RandSeed);
  SRAND ( RandSeed);
  PutHeadings;
  (* Generate MaxVals random integers *)
  FOR Index := 1 TO MaxVals DO
  BEGIN
    WRITE ( Index * 100 : 4, ':');
    Result := RAND MOD ( Index * 100);
    DispVariants ( Result);
  END; (* FOR Index := 1 TO MaxVals *)
  AwaitUser;
END.
```

The main program generates 10 pseudorandom **INTEGER**s. The first value is between 0 and 99, the second between 0 and 199, and so on up to the last value, which will be between 0 and 999. Output based on each of these values will be placed on a separate line.

In the main program, notice that the loop contains only one compound statement. In fact, the **FOR** construct applies to the statement immediately after the **FOR** loop specification. This can be a simple or compound statement. If it's a compound statement, the loop may carry out multiple instructions each time it executes. Together, the loop formulation and the statement following it make up the **FOR** statement.

The **FOR** loop in the main program consists of three actions grouped as a compound statement. Each time through the loop, the program carries out all three actions.

On each of the lines created in the main program, **DispVariants** writes five variants on the original pseudorandom value. Each variant is the sum of the starting value for the line and another pseudorandom integer. The first of these is between 0 and 1099, the second between 0 and 1199, and so on, until the last one, which is between 0 and 1499.

The **FOR** loop in **DispVariants** is followed by a simple statement. This is the only statement that is repeated in the procedure. The call to procedure **WRITELN** is made only after the **FOR** loop has repeated the required number of times.

Procedure **PutHeading** also contains two **FOR** loops. The first of these writes some headings for the table that the program generates. The second **FOR** loop draws a line by writing a single character over and over.

This program produces output such as the following:

```
      Val  1100 1200 1300 1400 1500
-------------
 100:   28  277  383  858  461  235
 200:   59  583 1205  754 1012 1212
 300:  268  873 1457 1506 1142  957
 400:  212 1174  932 1032  772  290
 500:   22  561  159 1085  541  536
 600:  196 1288 1317  932  672 1213
 700:  429 1093 1331  751  708  931
 800:  513  641  761 1080 1303 1212
 900:  352  837  773  488 1303 1702
1000:  449  721  463 1587 1461  893
```

The *loop variable*, also called the *control variable*, keeps track of the repetitions. It serves as a counter for the loop. This variable can be of any ordinal type. In the example program all loop variables happen to be **INTEGER**s; however, you'll see other types of loop variables later in the chapter and throughout the book.

The starting and ending values can be variables, values, or expressions, but must be of the same type as the loop variable. Thus, the **FOR** loop in the main program repeats as **Index** takes the values from 1 through **MaxValues** (10). The loop variable in **DispVariants** takes on values from 11 through **MaxVariants** (15).

There are no instructions that explicitly change the value of the loop variable. With a **FOR** loop, the program adjusts the value of the loop variable automatically. You don't need to change this value in your source code. In fact, it is very dangerous to do so.

While the **FOR** loop is executing, you should not explicitly change the value of the loop variable. Doing so will lead to problems. For example, a **FOR** loop such as the one in the following listing will lead to surprising results. Substitute the following listing for the loop in program **ForDemo1**, and compare the output with that from the original version of the program. The **Pascal Warnings** dialog box in the Project Settings command includes a setting that will warn you if your program explicitly changes the control variable in a **FOR** loop.

```
FOR Index := 1 TO MaxVals DO
BEGIN
  WRITE ( Index * 100 : 4, ':');
  Result := RANDOM ( Index * 100);
  DispVariants ( Result);
  INC ( Index);
END; (* FOR Index := 1 TO MaxVals *)
```

The value of the looping variable is undefined after the loop has finished. If you want to use the variable, you need to initialize it again.

If the ending value for a loop is smaller than the starting value, then the loop does not execute. If the ending value is equal to the starting value, then the loop executes once.

Do not put a semicolon after the **DO**. This would execute a null statement — that is, do nothing — and would end the **FOR** statement. To see what would happen, compile and run the following program. Watch the value of **Count** as you trace through the program.

Listing 5.2

```
(* Program to illustrate consequences of putting
   ; after DO
*)

PROGRAM BadForDemo;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

VAR Count : INTEGER;
BEGIN
  FOR Count := 1 TO 10 DO;
    WRITELN ( Count);
  WRITELN ( 'Done with FOR loop.');
  AwaitUser;
END.
```

This program writes only a single value of **Count**. The **WRITELN** statement is only executed once. The **FOR** loop, on the other hand, does nothing 10 times.

Looping Backwards

Suppose you want to write a sequence of letters in a reverse order — for instance, you may want to write a line such as

```
t    s    r    q    p    o    n    m    l    k    j
```

Is there an easy way to accomplish this?

In fact, there is another version of the **FOR** loop in which the loop variable counts backwards. The following program illustrates this version of the **FOR** loop. You may need to run the program a few times before it generates random values suitable for looping backwards.

Listing 5.3

```
(* Program to illustrate how to loop backwards with a
   FOR loop.
*)

PROGRAM ReverseLoop;
USES Windows;

VAR ChIndex, StartCh, EndCh : CHAR;
    RandSeed : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Generate a random character between 'a' and 'z' *)
  PROCEDURE RandomCh ( VAR TheCh : CHAR);
   (* Note that you can have an expression as the
     "value" of a constant.
  *)
  CONST Offset = ORD ( 'a');
  VAR Val : INTEGER;
  BEGIN
    (* Val will be between 97 ('a') and 122 ('z') *)
    Val := (RAND MOD 26) + Offset;
    TheCh := CHR ( Val);
  END;  (* RandomCh *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN (* Main *)
  GetInt ( 'Integer for random number generator', RandSeed);
  SRAND ( RandSeed);
  RandomCh ( StartCh);
  RandomCh ( EndCh);
  WRITE ( StartCh, ' to ', EndCh, ': ');
  (* Display a sequence of characters —
     from last down to first.
  *)
  FOR ChIndex := StartCh DOWNTO EndCh DO
    WRITE ( ChIndex : 2);
  WRITELN;
  AwaitUser;
END.
```

The program generates two random characters. The first is used as the starting character and the second as the ending character. The program then displays the letters between these two characters — from starting character down to the ending character. If the starting character comes earlier than the ending character, the loop does not execute. (Note, in **RandomCh**, that you can have an expression — **ORD('a')** — as the “value" of a constant.)

The only syntactic difference between the **FOR** loop in the **ReverseLoop** program and earlier **FOR** loops is the reserved word **DOWNTO** instead of **TO**. As a consequence of this, however, the value before the **DOWNTO** should be *greater* than the value after it.

If the final value is larger than the initial one, the **DOWNTO** version of the loop does not execute; if the two values are equal, the loop executes once.

Nested FOR Loops

You can put whatever kind of statement you want after a **FOR** loop specification — including a **FOR** statement. For example, suppose you wanted to write a display such as the following:

```
a.1  a.2  a.3  a.4  a.5  a.6  a.7  a.8  a.9
b.1  b.2  b.3  b.4  b.5  b.6  b.7  b.8  b.9
c.1  c.2  c.3  c.4  c.5  c.6  c.7  c.8  c.9
d.1  d.2  d.3  d.4  d.5  d.6  d.7  d.8  d.9
e.1  e.2  e.3  e.4  e.5  e.6  e.7  e.8  e.9
f.1  f.2  f.3  f.4  f.5  f.6  f.7  f.8  f.9
g.1  g.2  g.3  g.4  g.5  g.6  g.7  g.8  g.9
h.1  h.2  h.3  h.4  h.5  h.6  h.7  h.8  h.9
i.1  i.2  i.3  i.4  i.5  i.6  i.7  i.8  i.9
j.1  j.2  j.3  j.4  j.5  j.6  j.7  j.8  j.9
k.1  k.2  k.3  k.4  k.5  k.6  k.7  k.8  k.9
```

To get this display, you need to do the following:

- Write a line in which the same letter is repeated several times.
- For each repetition of the letter, pair a different digit with the letter, and display this pair, separated by a period.
- After the desired number of digits have been paired with the letter, move to the next line and the next letter.

The following program produces the output. Run the program to see what this output looks like. You might also want to trace through the two loops, with **Outer** and **Inner** as watch variables.

Listing 5.4

```
(* Program to illustrate use of nested FOR loops *)

PROGRAM NestedForDemo1;

CONST MaxSize = 9;
```

```
VAR Outer : CHAR;
    Inner : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  FOR Outer := 'a' TO 'k' DO
  BEGIN
    FOR Inner := 1 TO MaxSize DO
      WRITE  ( Outer:3, '.', Inner);
    WRITELN;
  END;
  AwaitUser;
END.
```

The **FOR** loop controlled by **Inner** is said to be *nested* within the loop controlled by **Outer**. This means that the loop controlled by **Inner** is executed for every value of **Outer**.

To develop your intuitions of how nested **FOR** loops behave, try making some variations in the preceding program. For example, try making the **WRITELN** part of the inner loop. You can do this in either of two ways, both of which will produce the same result.

```
(* combine WRITELN with WRITE *)
FOR Outer := 'a' TO 'k' DO
BEGIN
  FOR Inner := 1 TO MaxSize DO
    WRITELN  ( Outer:3, '.', Inner);
END;

(* make WRITELN part of the compound statement associated
   with inner loop.
*)
FOR Outer := 'a' TO 'k' DO
BEGIN
  FOR Inner := 1 TO MaxSize DO
  BEGIN
    WRITE  ( Outer:3, '.', Inner);
    WRITELN;
  END;
END;
```

Try using each of these loops in the **NestedForDemo1** program, to see how they alter the original version of the program.

You can also try to switch the specifications for the inner and outer loops. That is, you can try the **FOR** loop in the following program. Before you run the program, however, try to guess what the outcome will be.

Listing 5.5

```
(* Program to illustrate use of nested FOR loops *)

PROGRAM NestedForDemo2;

CONST MaxSize = 9;
VAR Outer : CHAR;
    Inner : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)
```

```
BEGIN
  FOR Inner := 1 TO MaxSize DO
  BEGIN
    FOR Outer := 'a' TO 'k' DO
      WRITE  ( Outer:3, '.', Inner);
    WRITELN;
  END;
  AwaitUser;
END.
```

When you run this version of the program, you'll find that the display looks as if the original output has been flipped. That is, the first row of the original output will be the first column of the new output, and the first column of the original will be the first row of the modified output.

When you have nested **FOR** loops, the inner loop will execute more often than the outer one. In fact, the entire inner loop will execute once for each pass through the outer loop.

You'll find **FOR** loops very useful in your work, and you'll find lots more examples using this construct.

Selection Constructs: The IF Statement

In your programs, you'll often want to carry out an action *if* a particular condition holds, but to proceed as usual if the condition does not hold.

For example, your program might search a collection of numbers, to see how often a particular value appears in the collection. The program will read each number in the collection and compare it with the value sought. If the current number equals the value sought, the program reports this fact; otherwise, the program simply continues (gets and compares the next number, and so forth).

Pascal's **IF** statement lets you direct program execution based on such a decision. The program generates pseudorandom integers. If the value is divisible by 10, the program writes a dot on the screen. After 50 dots, the program sends a **WRITELN** to display on the screen.

Listing 5.6

```
(* Program to illustrate use of IF construct.
   Program generates MaxTrials pseudorandom integers. If an
   integer is divisible by Divisor, the program writes a dot and
   increments a counter. After LineWidth dots have been written,
   the program writes the number of trials so far, and then
   moves to the next line.
*)

PROGRAM IfDemo;
USES Windows;

CONST MaxTrials = 5000;
      LineWidth = 50;
      Divisor = 10;
VAR Result, Index, NrHits, NrTrials, RandSeed : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN (* Main *)
  GetInt ( 'Integer for random number generator', RandSeed);
  SRAND ( RandSeed);
  NrHits := 0;
  NrTrials := 0;

  FOR Index := 1 TO MaxTrials DO
  BEGIN
    Result := RAND;
    INC ( NrTrials);
    (* If number is divisible by 10, write a dot, etc. *)
    IF ( Result MOD Divisor = 0) THEN
    BEGIN
      WRITE ( '.');
      INC ( NrHits);
      (* If there are enough dots on the line, ... *)
      IF ( NrHits MOD LineWidth = 0) THEN
        WRITELN ( NrTrials : 10);
    END; (* IF Result is divisible by 10 *)
  END; (* FOR Index := 1 to MaxTrials *)
  WRITELN;
  WRITELN ( MaxTrials, ' trials, ',
            NrHits, ' divisible by 10');
  AwaitUser;
END.
```

The expression in the **IF** statement must evaluate to a **BOOLEAN**. If this expression is **TRUE**, then the action or actions in the next statement are carried out. If the expression is **FALSE**, the program skips the next statement.

For example, in the program, the first **IF** statement tests whether the random value is divisible by 10. If so, the actions within the compound statement are carried out. If not, the program begins the next iteration of the **FOR** loop.

Note that the example contains a nested **IF** statement. This is allowed, just as nested **FOR** statements are allowed. In fact, notice that the first **IF** statement is nested inside the **FOR** loop. Any kind of control construct can be nested inside of any other kind. This simple property — known as *encapsulation* — is what enables you to build such varied and powerful programs.

TIP: The IF statement executes if the Boolean expression following the IF is TRUE. If so, the next simple or compound statement is executed; if not, the statement is bypassed.

Deciding Between Alternatives: The IF-ELSE Statement

Sometimes you'll want to decide between two alternative actions. If a particular condition holds, do one thing, otherwise do something different. To accomplish this, Pascal allows you to add an **ELSE** clause to an **IF** statement. This **IF-ELSE** variant enables you to do much more in your programs than a simple **IF** would allow.

The following program shows how to use the **IF-ELSE** construct.

Listing 5.7

```
(* Program to illustrate use of IF-ELSE construct *)

PROGRAM IfElseDemo;
USES Windows;

CONST MaxTrials = 500;
VAR Result, Count, HeadCount, RowCount, RandSeed : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)
```

```
BEGIN (* Main *)
  GetInt ( 'Integer for random number generator', RandSeed);
  SRAND ( RandSeed);
  HeadCount := 0;
  RowCount := 0;
  FOR Count := 1 TO MaxTrials DO
  BEGIN
    Result := RAND;

    (* If result is even, count it as a head,
       and write a period.
    *)
    IF result MOD 2 = 0 THEN
    BEGIN
      WRITE ( '.');
      RowCount := RowCount + 1;
    END
    ELSE   (* if not even, write a space *)
      WRITE ( ' ');

    (* after 50 trials, move to a new line *)
    IF Count MOD 50 = 0 THEN
    BEGIN
      WRITELN ( RowCount : 3);
      HeadCount := RowCount + HeadCount;
      RowCount := 0;
    END;  (* IF count MOD 50 = 0 *)
  END;
  WRITELN ( HeadCount, ' heads');
  AwaitUser;
END.
```

The program generates random integers. If the value is even, the program writes a dot, otherwise it writes a space. Every 50 trials, the program writes some summary information and then moves the cursor to the next line. The following listing shows output from a sample run:

```
.  ..  . .  .. .  ... .. .  .. ...  .... .... ...  30
.  .. .   ..   .. . .  . ..  .   .. ... . .    ..  23
.. ..  .. . . ..  ...    .. ...      . . . ... . . 26
 .  .  .. .  .. .   .    . ..   ..  .   ..  .  .   19
 ....      .   . .       . . .. . .  .  .. .  ..   19
.... ..   .........  .   ... . .. .    . ...    .  28
.... ... .       .   .      . . .  . . ....   .. . 22
  . .  .  .. ... . .. .     ..    .. .   .  .  .   20
. . ......  . ....   ...    .  . ... ...   . ..... 30
    . .  ..  . .....    .... ...    ...... . .   . 26
243 heads
```

The **ELSE** clause specifies the alternate action to take if the Boolean expression for the preceding **IF** clause is **FALSE**. The **ELSE** clause encompasses the simple or compound statement immediately after the **ELSE**.

Notice that there is no semicolon between the **END** that terminates the actions for the **IF** part and the **ELSE** that begins the alternative clause. This is one of the few places in Pascal where you *may not* have a semicolon.

Notice that there *are* semicolons between the statements *within* the compound statement preceding the **ELSE**. There is also a semicolon to separate the end of the **ELSE** clause (and, therefore, the end of the entire **IF-ELSE** statement) from subsequent source code.

TIP: You may not put a semicolon before an ELSE clause.

Syntactically, the entire **IF-ELSE** construct is treated as one statement by the compiler. If you were to put a semicolon before the **ELSE** you would separate the **ELSE** from the **IF**, and would make the **ELSE** clause a statement on its own. This is not possible, since the **ELSE** clause must have an **IF** clause preceding it.

Working Through Alternatives: An Extended IF-ELSE Statement

Sometimes you may have a whole sequence of decisions to make, with each decision depending on the answer to the previous one. The general format for such a series of decisions is **IF** *A* **THEN** <statement> **ELSE IF** *B* **THEN** <statement> **ELSE IF … THEN** <state-

ment> **ELSE** <statement>

You can have as many **ELSE IF** clauses as you need.

For example, consider the following simple problem, known as the *3A + 1 Problem*. The problem is defined in terms of the algorithm used to solve it. (For our purposes, an *algorithm* is simply a well-defined sequence of actions.)

Start with an arbitrary whole number, A.

1. If A = 1, stop.

2. Otherwise, if A is even, replace A with A DIV 2 and go back to step 2.

3. Otherwise, if A is odd, replace A with 3A + 1 and go back to step 2.

Interestingly enough, it's not known whether this algorithm will always end at 1 for any starting value. It *is* known that the algorithm stops for all values less than 536,870,912, or $2^{29)}$ The algorithm's behavior varies widely — even for numbers that are close to each other. For example, it takes 115 iterations to reach 1 when starting with 73, but only 22 iterations when starting with 74 and only 14 when starting with 75.

The following program applies the *3A + 1* algorithm to a value you specify, and also illustrates an extended **IF-ELSE** statement. Run the program with a starting value of 13.

Listing 5.8

```
(* Program to illustrate use of extended IF-ELSE statement.
   Program carries out 3A + 1 algorithm.
*)

PROGRAM ThreeADemo;

VAR Val, Count, Even, Odd : LONGINT;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetLongInt ( Message : STRING;
                         VAR Value : LONGINT);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetLongInt *)

  (* If a multiple of 20 lines have been written to the screen,
     pause to let user read them.
  *)
  PROCEDURE CheckForFullScreen ( NrLines : LONGINT);
  CONST FullPage = 20;
  BEGIN
    IF NrLines MOD FullPage = 0 THEN
    BEGIN
      WRITE ( 'Press ENTER to go on. ');
      READLN;
    END;
  END;  (* CheckForFullScreen *)

BEGIN
  WRITELN ( 'CTRL+C ends the program anytime.');
  Count := 0;
  Even := 0;
  Odd := 0;
  (* step 1 of the algorithm: get a starting value *)
  GetLongInt ( 'Value?', Val);
  (* the core of the algorithm *)
  REPEAT
    INC ( Count);
    (* step 2 in the algorithm *)
    IF ( Val = 1) THEN
      INC ( Odd)
    (* step 3 in the algorithm *)
    ELSE IF ( Val MOD 2 = 0) THEN
    BEGIN
      INC ( Even);
      Val := Val DIV 2;
    END
    (* if neither step 2 nor 3 applies, step 4 must apply *)
    ELSE  (* if Val is odd *)
    BEGIN
      INC ( Odd);
      Val := 3 * Val + 1;
    END;  (* IF-ELSE IF-ELSE Statement *)

    WRITELN ( 'Val = ', Val:8, '; Odd = ', Odd:8,
              '; Even = ', Even:8);
    CheckForFullScreen ( Count);
  UNTIL Val = 1;
  WRITELN ( 'After ', Count:8, ' iterations, result = 1.');
  WRITELN ( Even:8, ' even values; ', Odd:8, ' odd values');
  AwaitUser;
END.
```

You may want to work through an example, to get some experience with the extended **IF-ELSE** found in the program. The following listing shows the program's output for a starting value of 13. Execute the program by hand, to see how each intermediate value is computed.

```
Value? 13
Val =       40; Odd =        1; Even =        0
Val =       20; Odd =        1; Even =        1
Val =       10; Odd =        1; Even =        2
Val =        5; Odd =        1; Even =        3
Val =       16; Odd =        2; Even =        3
Val =        8; Odd =        2; Even =        4
Val =        4; Odd =        2; Even =        5
Val =        2; Odd =        2; Even =        6
Val =        1; Odd =        2; Even =        7
After        9 iterations, result = 1.
       7 even values;        2 odd values
```

You might also want to try some values of your own. If you end up picking a value that never seems to reach 1, press CTRL+C to break out of the program. Some values that end fairly quickly include 5, 10, 13, 16, or 21.

Several things are important to keep in mind about this extended **IF-ELSE** statement. First, notice that there are no semicolons before any of the **ELSE**s. This is because, syntactically, the entire sequence (beginning with the first **IF** and ending after the action for the last **ELSE**) is considered a single statement.

It's also important to keep in mind that all of the alternatives processed in this sequence are mutually exclusive — at least as far as the program is concerned. This means that *exactly one* of the clauses will be executed.

This exclusivity is the key to the extended **IF-ELSE** construct. Run the following version of the program, then look at the source code for the program. This version has removed the **ELSE** clauses, but has left the **IF** clauses. This version of the program will never terminate. To see why, consider what happens if a 1 reaches the third **IF**. You'll need to press CTRL+C when you are ready to stop this version of the program.

Listing 5.9

```
(* INCORRECT version of 3A + 1 algorithm.
   Extended IF-ELSE replaced by IF clauses.
*)

PROGRAM ThreeADemo;

VAR Val, Count, Even, Odd : LONGINT;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetLongInt ( Message : STRING;
                         VAR Value : LONGINT);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetLongInt *)

  (* If a multiple of 20 lines have been written to the screen,
     pause to let user read them.
  *)
  PROCEDURE CheckForFullScreen ( NrLines : LONGINT);
  CONST FullPage = 20;
  BEGIN
    IF NrLines MOD FullPage = 0 THEN
    BEGIN
      WRITE ( 'ENTER to go on; CTRL+C to stop.');
      READLN;
    END;
  END;  (* CheckForFullScreen *)

BEGIN
  WRITELN ( 'CTRL+C ends the program anytime.');
  Count := 0;
  Even := 0;
  Odd := 0;
  GetLongInt ( 'Value?', Val);
  (* the core of the algorithm *)
  REPEAT
    INC ( Count);
    IF ( Val = 1) THEN
      INC ( Odd);
    IF ( Val MOD 2 = 0) THEN
    BEGIN
      INC ( Even);
      Val := Val DIV 2;
    END;
    IF ( Val MOD 2 = 1) THEN
    BEGIN
      INC ( Odd);
      Val := 3 * Val + 1;
    END;
    WRITELN ( 'Val = ', Val:8, '; Odd = ', Odd:8,
              '; Even = ', Even:8);
    CheckForFullScreen ( Count);
  UNTIL Val = 1;
  WRITELN ( 'After ', Count:8, ' iterations, result = 1.');
  WRITELN ( Even:8, ' even values; ', Odd:8, ' odd values');
  AwaitUser;
END.
```

Notice that semicolons have been added after each of the **IF** clauses. These are necessary, since each **IF** clause in this program represents different statements.

Let's look at one more example comparing an extended **IF-ELSE** statement with several independent **IF** statements. The following program checks whether each in a series of numbers is divisible by 2, 3, 5, or 7. As soon as one of these values is found to divide the number, the program writes this information and moves on to the next number.

Listing 5.10

```
(* Program to illustrate extended IF-ELSE construct *)

PROGRAM DivisibilityDemo;

CONST Prime1 = 2;
      Prime2 = 3;
      Prime3 = 5;
      Prime4 = 7;
VAR Count, MaxTrials, StartValue, Result : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN
  GetInteger ( 'Starting value?', StartValue);
  GetInteger ( 'Maximum value? (> starting value)', MaxTrials);
  FOR Count := StartValue TO MaxTrials DO
  BEGIN
    WRITE ( Count:3, ': ');
    (* If divisible by Prime1 ... *)
    IF Count MOD Prime1 = 0 THEN
      WRITE ( Prime1:4)
    (* Else if divisible by Prime2 ... *)
    ELSE IF Count MOD Prime2 = 0 THEN
      WRITE ( Prime2:4)
    (* Else if divisible by Prime3 ... *)
    ELSE IF Count MOD Prime3 = 0 THEN
      WRITE ( Prime3:4)
    (* Else if divisible by Prime4 ... *)
    ELSE IF Count MOD Prime4 = 0 THEN
      WRITE ( Prime4:4)
    (* Else if not divisible by any of the previous values *)
    ELSE
      WRITE ( '**');
    WRITELN;
  END;
  AwaitUser;
END.
```

Because of the way the extended **IF-ELSE** works, exactly one of the clauses can be **TRUE**. As soon as one of the clauses satisfies its condition, the program stops processing the **IF-ELSE** statement. One consequence of this is that each number has exactly one divisor (or **) associated with it in the output, as in the following listing, which covers values from 2 through 25:

```
  2:    2
  3:    3
  4:    2
  5:    5
  6:    2
  7:    7
  8:    2
  9:    3
 10:    2
 11: **
 12:    2
 13: **
 14:    2
 15:    3
 16:    2
 17: **
 18:    2
 19: **
 20:    2
 21:    3
 22:    2
 23: **
 24:    2
 25:    5
```

Contrast this with the output from the following program, which converts the series of **ELSE IF** clauses into separate **IF** statements:

Listing 5.11

```
(* Program to illustrate separate IF statements.
   Compare program and output with DivisibiltiyDemo1.
 *)

PROGRAM DivisibilityDemo2;

CONST Prime1 = 2;
      Prime2 = 3;
      Prime3 = 5;
      Prime4 = 7;
VAR Count, StartValue, MaxTrials, Result : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN
  GetInteger ( 'Starting value?', StartValue);
  GetInteger ( 'Maximum value? (> starting value)', MaxTrials);
  FOR Count := StartValue TO MaxTrials DO
  BEGIN
    WRITE ( Count:3, ': ');
    (* If divisible by Prime1 ... *)
    IF Count MOD Prime1 = 0 THEN
      WRITE ( Prime1:4);
    (* if divisible by Prime2 ... *)
    IF Count MOD Prime2 = 0 THEN
```

```
      WRITE ( Prime2:4);
    (* if divisible by Prime3 ... *)
    IF Count MOD Prime3 = 0 THEN
      WRITE ( Prime3:4);
    (* if divisible by Prime4 ... *)
    IF Count MOD Prime4 = 0 THEN
      WRITE ( Prime4:4);
    (* Notice that the ELSE at the end has disappeared *)
    WRITE ( '**');
    WRITELN;
  END;
  AwaitUser;
END.
```

In this version, a particular value can have more than one prime divisor, since the divisors are checked independently of each other. This program produces the following output for values 2 through 25:

```
  2:    2**
  3:    3**
  4:    2**
  5:    5**
  6:    2   3**
  7:    7**
  8:    2**
  9:    3**
 10:    2   5**
 11: **
 12:    2   3**
 13: **
 14:    2   7**
 15:    3   5**
 16:    2**
 17: **
 18:    2   3**
 19: **
 20:    2   5**
 21:    3   7**
```

```
22:    2**
23: **
24:    2   3**
25:    5**
```

Beware of the Missing ELSE

Look at the following program before you run it. Try to determine what the output will be for an input of 0 and for an input of 5. Run the program for several different values — negative, 0, and positive. Remember which values you used, because you should run a related program with the same values. Trace through this program for these three values.

Listing 5.12

```
(* Program to illustrate dangling IFs *)

PROGRAM ElseTest;

VAR Val : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN
  GetInteger ( 'Value?', Val);
  IF Val <> 0 THEN
    IF ( Val < 0) THEN
      WRITELN ( 'Negative')
  (* according to indentation, this ELSE should be
     paired with the first IF.
     According to Pascal's syntax, however,
     it is paired with the inner IF.
  *)
  ELSE
      WRITELN ( 'Val = zero');
  AwaitUser;
END.
```

The program reads an integer value, and then responds differently, depending on whether **Val** is positive, negative, or zero. The program's indentation indicates the *intended* flow of control.

According to the indentation, the program tests whether **Val** is nonzero. If this is **TRUE**, the program should then test whether **Val** is negative. If so, the program writes “Negative." If **Val** is positive, the program is not supposed to do anything because no **ELSE** construct is intended after the inner **IF**. According to the indentation, the **IF** portion of the *outer* construct (**Val <> 0**) ends here. If the original test is **FALSE** — that is, if **Val** *is* zero — the program was intended to write “Val = zero" on the screen.

Unfortunately, the compiler sees this construct somewhat differently. It matches an **ELSE** with the *nearest* **IF** that does not have an **ELSE** associated with it. In practice, this is the *innermost* **IF** without an **ELSE**. In this case, the inner **IF** has no **ELSE** portion. Therefore, the compiler associates the **ELSE** with this **IF**.

The result is that the program writes “Val = zero" when **Val** has the value 5. Because the **ELSE** is associated with the inner **IF**, there is no alternative action for the outer **IF**. This means that the program does nothing when **Val** has the value 0.

There are two ways around this difficulty. First, you can include an inner **ELSE** that simply tells the program to do nothing, as in the next program. Run this program, using the same input as for the previous program. Compare the output from the two programs.

Listing 5.13

```
(* Program to illustrate a fix for dangling IFs *)

PROGRAM ElseTest2;

VAR Val : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN
  GetInteger ( 'Value?', Val);
  IF Val <> 0 THEN
    IF ( Val < 0) THEN
      WRITELN ( 'Negative')
     (* The following ELSE is an instruction to do
        nothing, but serves to close up the inner IF.
        Note that no semicolon follows the new ELSE.
    *)
    ELSE  (* Do nothing *)
  ELSE
      WRITELN ( 'Val = zero');
  AwaitUser;
END.
```

The second way of dealing with the missing **ELSE** problem is to use **BEGIN** and **END** to create a compound statement — thereby grouping the statements the way you want them. The following listing shows how to do this:

Listing 5.14

```
(* Program to illustrate a fix for dangling IFs *)

PROGRAM ElseTest3;

VAR Val : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN
  GetInteger ( 'Value?', Val);
  IF Val <> 0 THEN
```

```
   (* Put the inner IF within a compound statement *)
  BEGIN
    IF ( Val < 0) THEN
      WRITELN ( 'Negative')
  END  (*  compound statement that's executed if Val <> 0 *)
  ELSE
      WRITELN ( 'Val = zero');
  AwaitUser;
END.
```

In this version, the first **IF** is followed by a compound statement. This statement contains the inner **IF**. The **END** that terminates the compound statement also serves to separate the inner **IF** from the **ELSE**. In this way, the **ELSE** is grouped with the starting **IF**, as intended.

The three versions of the **IF** construct — **IF**, IF-ELSE, and extended **IF-ELSE** — represent the first type of selection construct. You'll find **IF** statements of various sorts throughout the book, and you'll rarely write a program in which you don't use at least one **IF** statement.

Selecting from Among Several Possibilities: The CASE Statement

Pascal has another selection construct that is particularly useful when you have numerous things to test for, or when a test involves lots of possibilities. For example, to test whether a particular character is a vowel, you might be tempted to use the following (very long) Boolean expression:

```
IF ( TestCh = 'a') OR ( TestCh = 'e') OR ( TestCh = 'i') OR
   ( TestCh = 'o') OR ( TestCh = 'u') OR ( TestCh = 'A') OR
   ( TestCh = 'E') OR ( TestCh = 'I') OR ( TestCh = 'O') OR
   ( TestCh = 'U') THEN
```

Or, suppose you want to test whether a character is a vowel, a consonant, an even digit, an odd digit, or another printable character. One approach to this would be through an extended **IF-ELSE** statement. The structure of such a test would be: **IF TestCh** is a consonant **THEN … ELSE IF TestCh** is a vowel **THEN … ELSE IF TestCh** is an even digit **THEN … ELSE IF TestCh** is an odd digit **THEN … ELSE …**

Each of the individual tests — such as whether **TestCh** is a consonant or whether it's an even digit — can be long Boolean expressions.

Pascal's **CASE** statement makes it much easier to test for conditions such as these, and CodeWarrior Pascal's extensions of the **CASE** statement make your job even easier. The **CASE** statement provides selection criteria based on the value of a particular test variable or expression, which must be of an ordinal type. Essentially, the **CASE** statement allows you to group and list all the values of the variable or expression that lead to each of your possible actions.

For example, the following program lets you test whether a particular character falls into any of the groups discussed previously — consonant, vowel, and so forth.

Listing 5.15

```
(* Program to illustrate use of CASE statement *)

PROGRAM CaseDemo;

VAR TestStr : STRING;
    Index, HowLong, NrConsonants, NrVowels,
    NrEven, NrOdd, NrPrintable : INTEGER;
    Ch : CHAR;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

BEGIN
  GetString ( 'String to process?', TestStr);
  HowLong := LENGTH ( TestStr);
  NrConsonants := 0;
  NrVowels := 0;
  NrEven := 0;
  NrOdd := 0;
  NrPrintable := 0;
  FOR Index := 1 TO HowLong DO
  BEGIN
    Ch := UPCASE ( TestStr [ Index]);
    (* Use Ch as the basis for decision making *)
    CASE Ch OF
      'A', 'E', 'I', 'O', 'U' : INC ( NrVowels);
      'B' .. 'D', 'F' .. 'H',
      'J' .. 'N', 'P' .. 'T', 'V' .. 'Z' : INC ( NrConsonants);
      '0', '2', '4', '6', '8' : INC ( NrEven);
      '1', '3', '5', '7', '9' : INC ( NrOdd);
      OTHERWISE  INC ( NrPrintable);
    END; (* CASE of TestStr [ Index] *)
  END;  (* FOR Index := 1 TO HowLong *)
  WRITELN ( 'NrVowels    = ', NrVowels:3,
            '; NrConsonants = ', NrConsonants:3);
  WRITELN ( 'NrEven      = ', NrEven:3,
            '; NrOdd        = ', NrOdd:3);
  WRITELN ( 'NrPrintable = ', NrPrintable:3);
  AwaitUser;
END.
```

The program reads a string, and then looks at each character. Depending on the category to which the character belongs, a particular counter is incremented.

LENGTH is a predefined function in CodeWarrior Pascal, which returns the number of characters in the string argument. **TestStr [Index]** specifies the **Index**th character in the string. The **UPCASE** function converts its character argument to uppercase — if the character is a letter — and returns the uppercase version.

The **CASE** statement works as follows:

- Each time through the **CASE** statement, the value of **Ch** is checked.
- If this value is a vowel — that is, has one of the five values listed (`A' through `U') — then **NrVowels** is increased by 1. The statement (simple or compound) following the colon (:) is executed. If this happens, processing of the **CASE** statement is complete, and the program continues with the next iteration of the **FOR** loop.
- If the value is not a vowel, the program checks the next set of candidate values — which represent the consonants. These are specified as characters in several consecutive clusters. For example, all characters between `B' and `D' (that is, `B,' `C,' and `D') are consonants, as are all characters between (for example) `J' and `N.' Syntactically, such a range of characters is indicated by two dots (..) between the starting and ending values. If the value of **Ch** is one of these, then **NrConsonants** is increased by 1, and the **CASE** statement is finished.
- If the character is neither vowel nor consonant, the program checks whether the character is an even digit, and then whether it is an odd digit.
- If the value of **Ch** still has not been characterized at this point, the **OTHERWISE** clause of the **CASE** statement is applied. In this case, **NrPrintable** is increased by 1. Notice that there is no colon between the **OTHERWISE** and the statement executed. Notice also that an **END** terminates the **CASE** statement. This is one of three places in Pascal in which you find an **END** without a corresponding **BEGIN**.

The structure of a **CASE** statement is: **CASE** <ordinal value or expression> **OF** <value list 1> **:** <simple or compound statement>**;** <value list 2> **:** < simple or compound statement>**;** … <value list n> **:** <simple or compound statement>**; OTHERWISE** <simple or compound statement> **END;**

The line beginning with **OTHERWISE** is optional, and is allowed in CodeWarrior Pascal, but not in standard Pascal. You can have at most one **OTHERWISE** clause in a **CASE** statement.

The **CASE** statement is controlled by the variable or expression following the word **CASE**. This will generally be a variable that can take on the values covered in the individual components of the **CASE** statement. It may also be an expression (such as **MyInt** + 7) that will evaluate to one of the values. Syntactically, it could even be an actual value (such as 7), but you wouldn't really need a **CASE** statement in such a situation.

A *value list* may consist of a single value, several values separated by commas, or of a range of values indicated by a starting value, two dots, and an ending value. The ability to specify a range of value is an extension to standard Pascal, and is available in CodeWarrior Pascal.

When the program encounters a **CASE** statement, the value lists are searched until a match is found or until the end of the **CASE** statement. When a match is found, the statement associated with the appropriate value list is executed. After this, processing of the **CASE** statement is finished. If none of the specified values match, and you have an **OTHERWISE** clause in the **CASE** statement, then the statement associated with the **OTHERWISE** is executed.

If there are no matches and you do not have an **OTHERWISE** clause, then a CodeWarrior Pascal program does nothing inside the **CASE** statement, and execution continues with the statement following the **CASE** statement. If such a situation arises in standard Pascal, the program's behavior is undefined.

As usual, the statement associated with a value list can be a simple or a compound one. In fact, you can even have another **CASE** statement associated with a value list — that is, you can have a **CASE** statement nested inside the specification for another such statement.

The following program illustrates a nested **CASE** statement. Before running the program, work through an iteration of the **FOR** loop in the main program for **Count** values of 8 and 9. You should trace through at least one example involving a **CASE** statement. This is a good candidate.

Listing 5.16

```
(* Program to illustrate nested CASE statements *)

PROGRAM NestedCaseDemo;
USES Windows;

CONST MaxTrials = 10;
VAR Count, Value, RandSeed : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)
  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN (* Main *)
  GetInt ( 'Integer for random number generator', RandSeed);
  SRAND ( RandSeed);
  FOR Count := 1 TO MaxTrials DO
  BEGIN
    Value := RAND;
    (* Notice that the CASE statement also can be controlled
       by an expression.
    *)
    CASE  Value MOD 4 OF
      0 :
        BEGIN
          WRITELN ( Value:5, ' is divisible by 4');
          Value := RAND;
          (* A nested CASE statement — fired only if the outer
             result was 0. Notice that the same variable can
             be involved in both CASE statements.
          *)
          CASE Value MOD 2 OF
            0 : WRITELN ( Value:15, ' is even');
            1 : WRITELN ( Value:15, ' is odd');
          END; (* Nested CASE of Value *)
        END;
      1 : WRITELN ( Value:5, ': Remainder is 1');
      2 : WRITELN ( Value:5, ': Remainder is 2');
      3 : WRITELN ( Value:5, ': Remainder is 3');
      (* OTHERWISE clause is optional, and is not needed here *)
    END;   (* CASE of Value MOD 4 *)
  END; (* FOR Count := 1 TO MaxVals *)
  AwaitUser;
END.
```

The program generates at least 10 random integer values, and displays some information about these values. If a generated value is divisible by 4, the program generates another random value and checks whether this value is odd or even. This second test is done in a **CASE** statement nested within one of the possible values for the outer **CASE** statement. The first **END** under the 0 case terminates the nested **CASE** statement; the second one terminates the compound statement associated with the 0.

For a sample run, the program produced the following output:

```
16382: Remainder is 2
18067: Remainder is 3
27128 is divisible by 4
          21344 is even
  726: Remainder is 2
 1340 is divisible by 4
          11832 is even
15911: Remainder is 3
21374: Remainder is 2
 6027: Remainder is 3
 6816 is divisible by 4
          11935 is odd
17031: Remainder is 3
```

Notice that both of the **CASE** statements in this program are controlled by expressions (rather than variables) that evaluate to a small range of possible integer values.

Iteration Constructs

Earlier in this chapter, you learned about the **FOR** loop, which repeats something a specified number of times. The loop is controlled by a counter that is updated automatically by the program. When the counter reaches the specified ending value, the loop stops executing.

More commonly, your program loops will need to execute an indeterminate number of times. The criterion for stopping or continuing the loop will be some condition, which will be tested before or after each time through the loop (that is, each iteration).

For example, suppose you wanted to determine how long it would take you, on average, to generate a random value divisible by 17. You might start generating random integers, and counting the number of values generated up to that point. You would continue generating values until you got one that was divisible by 17.

The Impulsive Iteration Construct: The REPEAT *Statement*

The following program lets you run multiple trials of this sort, and introduces an iteration construct for controlling such a loop. Run the program a few times. Do the results seem to be clustered within a particular range? If so, can you think of an explanation for this result?

Listing 5.17

```
(* Program to illustrate use of REPEAT construct *)

PROGRAM RepeatDemo;
USES Windows;

CONST Divisor = 17;
      MaxTrials = 1000;
VAR Value, Count, NrTries,
    RunningSum, ShortestRun, LongestRun : LONGINT;
    RandSeed : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE CheckExtremes ( Value : LONGINT;
                            VAR Shortest, Longest : LONGINT);
  BEGIN
    IF Value > Longest THEN
      Longest := Value;
    IF Value < Shortest THEN
      Shortest := Value;
  END; (* CheckExtremes *)

  PROCEDURE DisplaySummary ( NrTrials, Total,
                             Shortest, Longest : LONGINT);
  BEGIN
    WRITELN ( 'Longest Run = ', Longest:3);
    WRITELN ( 'Shortest Run = ', Shortest:3);
    WRITELN ( 'Average Run Length = ',
              Total / NrTrials : 10 : 3);
  END; (* DisplaySummary *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN (* Main *)
  GetInt ( 'Integer for random number generator', RandSeed);
  SRAND ( RandSeed);
  RunningSum := 0;
  ShortestRun := MAXINT;
  LongestRun := 0;

  WRITELN ( 'Generating values ...');
  FOR Count := 1 TO MaxTrials DO
  BEGIN
    NrTries := 0;
    (* Repeat the following loop until program
       generates a number that's divisible by Divisor.
       This loop executes AT LEAST once.
       The REPEAT loop ends with the UNTIL line.
    *)
    REPEAT
      Value := RAND;
      INC ( NrTries);
    UNTIL ( Value MOD Divisor = 0);
    CheckExtremes ( NrTries, ShortestRun, LongestRun);
    RunningSum := RunningSum + NrTries;
  END; (* FOR Count := 1 TO MaxTrials *)

  DisplaySummary ( MaxTrials, RunningSum,
                   ShortestRun, LongestRun);
  AwaitUser;
END.
```

This program is controlled by the **REPEAT** loop nested inside the **FOR** loop. Such a loop keeps executing a series of statements until some condition becomes **TRUE**. This test condition is known as the *termination condition*, since the loop must repeat until the termination condition becomes **TRUE**.

For a few sample runs, this program produced the following results:

```
Longest Run = 100
Shortest Run = 1
Average Run Length =     17.451

Longest Run = 124
Shortest Run = 1
Average Run Length =     17.112

Longest Run = 99
Shortest Run = 1
Average Run Length =     16.428

Longest Run = 134
Shortest Run = 1
Average Run Length =     17.055

Longest Run = 153
Shortest Run = 1
Average Run Length =     16.833
```

Change the value of **MaxTrials** to 100 in program **RepeatDemo**. Then run several new samples. Do the results from the second series of runs differ more dramatically from each other than when you ran 1000 trials each time? This increased variability (range of values) in the outcome occurs because a smaller number of trials gives you a less stable result.

Finally, change the value of **Divisor**, then run the program again. Do the results cluster around a different value this time? The average run length is directly related to the divisor. For example, if the divisor were 2, you would expect to get an even number about every other time. This would mean that, on the average, it would take a run length of 2 to give you the desired result.

The structure of a **REPEAT** loop is: **REPEAT** <any sequence of simple of compound statements> **UNTIL** <a Boolean expression becomes **TRUE**>. The Boolean expression used to make a decision is called the *termination condition.*

Notice that you don't need a **BEGIN** and **END** to delimit the statements that make up the **REPEAT** loop. This is accomplished by the **REPEAT** at the start and the **UNTIL** at the end of the loop.

A **REPEAT** loop does its work first, and then tests whether it can stop. This execution format — act, then test — means that the loop will execute *at least* once, since it doesn't test until *after* it has done its work.

The termination condition will generally involve a particular variable, whose value determines whether the Boolean expression is **TRUE** or **FALSE**. This variable is called the *loop variable*. In order for the loop to terminate eventually, the value of the loop variable needs to change over iterations. If the loop variable stayed constant, and its initial value did not make the termination condition **TRUE**, then the loop would never stop executing.

For example, program **RepeatDemo** depends on the fact that you'll eventually generate a random number divisible by **Divisor**. While there is no absolute guarantee of this, you can be very confident that such a number will be generated.

In a **REPEAT** loop, it's *your* responsibility to make sure that the value of the loop variable changes. You can accomplish this by generating values (as in the example program) or by assigning different values to the loop variable. The **REPEAT** loop differs from the **FOR** loop in that *you* must make such changes or must tell your program to make them. Recall that in a **FOR** loop the program updates the loop (that is, counter) variable automatically.

TIP: A REPEAT loop carries out its task and *then* checks whether it can stop. For this reason such a loop executes at least once. It's your responsibility to make sure the value of

the loop variable changes over iterations, so that the termination condition can eventually become TRUE.

The Cautious Iteration Construct: The WHILE Statement

In contrast to the **REPEAT** statement, which acts first, and asks questions later, Pascal's **WHILE** construct checks first, and acts only if certain conditions are fulfilled (that is, are **TRUE**). You've already seen the **WHILE** construct in earlier programs. Its syntax is: **WHILE** <the Boolean expression evaluates to **TRUE**> **DO** <simple or compound statement>.

The Boolean expression in this statement is called the *continuation condition*. A **WHILE** statement executes *while* the continuation condition is **TRUE**. Thus, a **WHILE** loop stops when a condition no longer holds — that is, becomes **FALSE**; in contrast, a **REPEAT** loop stops when a condition becomes **TRUE**.

The next program also looks for a number divisible by 17, but uses a **WHILE** statement.

Listing 5.18

```
(* Program to illustrate use of WHILE construct *)

PROGRAM WhileDemo;
USES Windows;

CONST Divisor = 17;
      MaxTrials = 1000;
VAR Value, Count, NrTries,
    RunningSum, ShortestRun, LongestRun : LONGINT;
    RandSeed : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE CheckExtremes ( Value : LONGINT;
                            VAR Shortest, Longest : LONGINT);
  BEGIN
    IF Value > Longest THEN
      Longest := Value;
    IF Value < Shortest THEN
      Shortest := Value;
  END; (* CheckExtremes *)

  PROCEDURE DisplaySummary ( NrTrials, Total,
                             Shortest, Longest : LONGINT);
  BEGIN
    WRITELN ( 'Longest Run = ', Longest:3);
    WRITELN ( 'Shortest Run = ', Shortest:3);
    WRITELN ( 'Average Run Length = ',
              Total / NrTrials : 10 : 3);
  END; (* DisplaySummary *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN (* Main *)
  GetInt ( 'Integer for random number generator', RandSeed);
  SRAND ( RandSeed);
  RunningSum := 0;
  ShortestRun := MAXINT;
  LongestRun := 0;
  WRITELN ( 'Generating values ...');
  FOR Count := 1 TO MaxTrials DO
  BEGIN
    Value := RAND;
    NrTries := 1;
    (* As long as the most recently generated value
       is NOT divisible by Divisor, execute the loop.
       If the first value generated is divisible by Divisor,
       this loop will never execute.
       Compare the Boolean expression with the one for the
```

```
        REPEAT loop.
     *)
     WHILE ( Value MOD Divisor <> 0) DO
     BEGIN
       Value := RAND;
       INC ( NrTries);
     END;
     CheckExtremes ( NrTries, ShortestRun, LongestRun);
     RunningSum := RunningSum + NrTries;
   END; (* FOR Count := 1 TO MaxTrials *)
   DisplaySummary ( MaxTrials, RunningSum,
                    ShortestRun, LongestRun);
   AwaitUser;
 END.
```

This program does exactly the same things as the previous one. However, the inner looping is controlled by a **WHILE** instead of a **REPEAT** loop. This has several consequences.

Because the **WHILE** loop tests before doing anything, it's necessary to generate a value outside the loop. This value primes the loop, so that the loop variable in the continuation condition has been initialized. This is not necessary with the **REPEAT** loop because it initializes the loop variable in the body of the loop, which comes before the test.

Because you're looking for a **FALSE** result in a **WHILE** loop, the Boolean expression in the program looks for values that are *not* divisible by **Divisor**. This test is essentially the opposite of the test made in the **REPEAT** loop in an earlier example.

As it turns out, the **REPEAT** and **WHILE** statements can be made equivalent to each other. In principle, this means the constructs are interchangeable; wherever you can use one type of loop, you can also use the other. However, in particular situations, one or the other type of loop will generally be more convenient.

Because the continuation condition could be **FALSE** on the very first test, the body of the **WHILE** loop may never execute.

As do the other control constructs — except for the **REPEAT** construct — the **WHILE** applies to the next statement, simple or compound.

The program will not change a **WHILE** statement's loop variable automatically. As a consequence, you need to change this variable — either explicitly through assignment or through procedure or function calls. In practice, you'll almost always have a compound statement for a **WHILE** statement. For a **WHILE** loop, you *will* need a **BEGIN** and **END** to mark the boundaries of the statement.

TIP: A WHILE loop checks whether it can continue, and carries out its task *only* if it can. For this reason such a loop may never execute. It's your responsibility to make sure the value of the loop variable changes over iterations, so that the continuation condition can eventually become FALSE.

To illustrate the importance of changing the loop variable, consider the following program, which generates a table of the printable ASCII characters.

Listing 5.19

```
(* Program to illustrate WHILE loop, and
   importance of changing value of loop variable.
*)

PROGRAM ASCIIDisplay;

VAR Count : INTEGER;
    CurrCh : CHAR;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Display the current character;
      Move to a new line after every 5 values.
  *)
  PROCEDURE Display;
  BEGIN
    WRITE ( '     ', ORD ( CurrCh) : 3, '  ', CurrCh);
    INC ( Count);
    IF Count MOD 5 = 0 THEN
      WRITELN;
  END;

BEGIN
  Count := 0;
  (* space — #32 — is the first printable CHAR *)
  CurrCh := #32;
  (* #126 is the last printable character *)
  WHILE ORD ( CurrCh) < 127 DO
  BEGIN
    Display;
    (* get the next character *)
    CurrCh := SUCC ( CurrCh);
  END;
  AwaitUser;
END.
```

The program does two things as long as **CurrCh** is less than 127: display the character and increase the value of **CurrCh** within the loop. This program produces the following output:

```
      32         33  !      34  "      35  #      36  $
      37  %      38  &      39  '      40  (      41  )
      42  *      43  +      44  ,      45  -      46  .
      47  /      48  0      49  1      50  2      51  3
      52  4      53  5      54  6      55  7      56  8
      57  9      58  :      59  ;      60  <      61  =
      62  >      63  ?      64  @      65  A      66  B
      67  C      68  D      69  E      70  F      71  G
      72  H      73  I      74  J      75  K      76  L
      77  M      78  N      79  O      80  P      81  Q
      82  R      83  S      84  T      85  U      86  V
      87  W      88  X      89  Y      90  Z      91  [
 92  \      93  ]      94  ^      95  _      96  `
 97  a      98  b      99  c     100  d     101  e
102  f     103  g     104  h     105  i     106  j
107  k     108  l     109  m     110  n     111  o
112  p     113  q     114  r     115  s     116  t
117  u     118  v     119  w     120  x     121  y
122  z     123  {     124  |     125  }     126  ~
```

If you remove the line

```
CurrCh := SUCC ( CurrCh);
```

the program will never terminate, since it will display a space (ASCII 32) over and over.

The Structure of Iteration Statements

Although their syntax is quite simple, iteration statements can have subtle effects. The output from a **WHILE** or **REPEAT** loop can vary quite a bit, depending on the details of the Boolean expression that controls the loop. In particular, three things affect the way in which an iteration loop does its work:

- The boundary (starting and ending) values for the loop variable.
- The actual test being done in the Boolean expression (continuation or termination condition).
- Where in the loop body the loop variable is changed.

The following example illustrates how these factors affect a loop. The program computes and displays some numerical values. Read the program and then run it, to see what the program's routines do.

Listing 5.20

```
(* Program to illustrate effects of varying details
   of a WHILE loop.
*)

PROGRAM LoopDemo;

CONST MaxValue = 10;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Return the cube of the argument *)
  FUNCTION Cube ( Val : INTEGER) : INTEGER;
  BEGIN
    Cube := Val * Val * Val;
  END; (* FN Cube *)

   (* Compute and display a series of cubes;
      change loop variable BEFORE computing and
      displaying results.
  *)
  PROCEDURE WChangeAtTop;
  VAR Count, RunningSum, Result : INTEGER;
  BEGIN
    Count := 1;
    RunningSum := 0;
    WHILE Count <= MaxValue DO
    BEGIN
      INC ( Count);
      Result := Cube ( Count);
      RunningSum := RunningSum + Result;
      WRITELN ( Count:2, ': Cube = ', Result:4,
                '; RunningSum = ', RunningSum:4);
    END; (* WHILE Count <= MaxValue *)
  END; (* WChangeAtTop *)

  (* Compute and display a series of cubes;
     change loop variable AFTER computing and
     displaying results.
  *)
  PROCEDURE WChangeAtBottom;
  VAR Count, RunningSum, Result : INTEGER;
  BEGIN
    Count := 1;
    RunningSum := 0;
    WHILE Count <= MaxValue DO
    BEGIN
      Result := Cube ( Count);
      RunningSum := RunningSum + Result;
      WRITELN ( Count:2, ': Cube = ', Result:4,
                '; RunningSum = ', RunningSum:4);
      INC ( Count);
    END; (* WHILE Count <= MaxValue *)
  END; (* WChangeAtBottom *)

BEGIN
  WRITELN ( 'WHILE ChangeAtBottom:');
  WChangeAtBottom;
  WRITELN ( '———————————————');
  WRITELN ( 'WHILE ChangeAtTop:');
  WChangeAtTop;
  AwaitUser;
END.
```

This program produces the following output. You should work through these two procedures by hand — to help develop your understanding of how they behave. Then trace through the program to confirm your hand execution.

```
WHILE ChangeAtBottom:
 1: Cube =    1; RunningSum =    1
 2: Cube =    8; RunningSum =    9
 3: Cube =   27; RunningSum =   36
 4: Cube =   64; RunningSum =  100
 5: Cube =  125; RunningSum =  225
 6: Cube =  216; RunningSum =  441
 7: Cube =  343; RunningSum =  784
 8: Cube =  512; RunningSum = 1296
 9: Cube =  729; RunningSum = 2025
10: Cube = 1000; RunningSum = 3025
——————————————
WHILE ChangeAtTop:
 2: Cube =    8; RunningSum =    8
 3: Cube =   27; RunningSum =   35
 4: Cube =   64; RunningSum =   99
 5: Cube =  125; RunningSum =  224
 6: Cube =  216; RunningSum =  440
 7: Cube =  343; RunningSum =  783
 8: Cube =  512; RunningSum = 1295
 9: Cube =  729; RunningSum = 2024
10: Cube = 1000; RunningSum = 3024
11: Cube = 1331; RunningSum = 4355
```

The only difference between **WChangeAtTop** and **WChangeAtBottom** is the location of the statement:

```
INC ( Count);
```

In **WChangeAtTop**, the loop variable's value is changed *before* anything is done with it in the loop. In this case, the loop works with the values 2 through 11, instead of 1 through 10. On the other hand, by changing this variable at the end of the loop body — as in

WChangeAtBottom — the procedure works with the values 1 through 10.

You could use **REPEAT** loops in the procedures. In that case, the termination condition corresponding to the continuation conditions in the current program would be:

```
Val > MaxValue
```

To explore the effects of changes in the factors mentioned earlier on the behavior of a loop, display the sums of cubes information for the condition combinations summarized in Table 5-1:

Table 6: Suggested Variants on the WHILE and REPEAT Loops

Initial Looping Variable	Continuation Condition	Change in Looping Variable
1	Count <= MaxValue	Bottom
1	Count <= MaxValue	Top
1	Count < MaxValue	Bottom
1	Count < MaxValue	Top
0	Count <= MaxValue	Bottom
0	Count <= MaxValue	Top
0	Count < MaxValue	Bottom
0	Count < MaxValue	Top

A Comparison Example

The following program lets you compare the behavior of **WHILE** and **REPEAT** loops directly. In particular, you can see what the consequences are of testing before or after the loop.

Listing 5.21

```
(* Program to enable direct comparison between
   WHILE and REPEAT loops.
*)

PROGRAM WhileAndRepeatDemo;

VAR Value : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

  PROCEDURE WhileTest ( Count : INTEGER);
  BEGIN
    WHILE ODD ( Count) DO
    BEGIN
      WRITE ( Count : 0, ' : ');
      Count := Count DIV 2;
    END;
    WRITELN ( 'Done');
  END; (* WhileTest *)

  PROCEDURE RepeatTest ( Count : INTEGER);
  BEGIN
    REPEAT
      WRITE ( Count : 0, ' : ');
      Count := Count DIV 2;
    UNTIL ( NOT ( ODD ( Count)));
    WRITELN ( 'Done');
  END; (* RepeatTest *)

BEGIN
  REPEAT
    GetInteger ( 'Value? (0 to stop) ', Value);

    WRITELN ( 'WhileTest:');
    WhileTest ( Value);
    WRITELN ( '————————--');
    WRITELN ( 'RepeatTest:');
    RepeatTest ( Value);
  UNTIL ( Value = 0);
  AwaitUser;
END.
```

The following listing shows some sample outputs for selected starting values. Compare the behavior of the **WHILE** and the **REPEAT** loops.

```
Value? 14
WhileTest:
Done
————————--
RepeatTest:
14 : 7 : 3 : 1 : Done

Value? 37
WhileTest:
37 : Done
————————--
RepeatTest:
37 : Done

Value? 31
WhileTest:
31 : 15 : 7 : 3 : 1 : Done
————————--
RepeatTest:
31 : 15 : 7 : 3 : 1 : Done
```

```
Value? 64
WhileTest:
Done
----------
RepeatTest:
64 : Done
```

Notice that, for certain values, the **WHILE** loop never computes any results, whereas the **REPEAT** loop always displays at least one value.

Getting Around the Looping Rules

CodeWarrior Pascal includes several predefined routines that let you take special action during a loop. The four procedures — **CYCLE**, **EXIT**, **HALT**, and **LEAVE** — all alter the flow of control in a program.

CYCLE and **LEAVE** must appear within a **FOR**, **WHILE**, or **REPEAT** loop; the other two can appear anywhere within a routine. Practically speaking, **CYCLE** and **LEAVE** will always appear in a compound statement associated with a looping construct. This means there will probably be program statements before and after calls to these procedures.

CYCLE

The **CYCLE** procedure is the least disruptive of the group. This procedure causes the next iteration of the loop to begin immediately. Any statements in the loop's compound statement that follow the call to **CYCLE** will be skipped, since the current iteration ends before they're executed again. The statements will execute normally in the next iteration — unless, of course, **CYCLE** is called again.

The following program illustrates the use of the **CYCLE** routine. The program writes all values between 1 and 15,000 that are divisible by each of the first prime numbers: 2, 3, 5, 7, and 11. The strategy in the program is to check each of the divisors in succession.

Such a program will be much faster if it can stop processing a value as soon as the value fails a divisibility test. The example program uses **CYCLE** to short-circuit the testing process when appropriate. By doing this, the program saves almost 50,000 divisibility tests.

Listing 5.22

```
(* Program to illustrate use of CYCLE procedure.
   Program finds all numbers between 1 and Target that
   are divisible by the first six prime numbers.
*)

PROGRAM CYCLETest;

CONST Target = 15000;
VAR Count : INTEGER;

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  Count := 0;
  WHILE Count <= Target DO
  BEGIN
    INC ( Count);
    (* as soon as a divisor is found, CYCLE to next value *)
    IF ( Count MOD 2) <> 0 THEN
      CYCLE
    ELSE IF ( Count MOD 3) <> 0 THEN
      CYCLE
    ELSE IF ( Count MOD 5) <> 0 THEN
      CYCLE
    ELSE IF ( Count MOD 7) <> 0 THEN
      CYCLE
    ELSE IF ( Count MOD 11) <> 0 THEN
      CYCLE
    ELSE (* if divisible by 2, 3, 5, 7, and 11 *)
      WRITELN ( Count);
  END; (* WHILE Count <= Target *)
  WRITELN ( 'Upon exit, Count = ', Count);
  AwaitUser;
END.
```

When you run the program, it produces the following output:

```
    2310
    4620
    6930
    9240
   11550
   13860
Upon exit, Count =    15001
Press ENTER to continue
```

LEAVE

LEAVE immediately ends the loop within which the routine is called — so that there are no more iterations. Control is transferred immediately to the first statement *after* the loop. To see the difference between **CYCLE** and **LEAVE**, change the calls in the preceding program from **CYCLE** to **LEAVE**, as in the following listing.

Listing 5.23

```
(* Program to illustrate use of LEAVE procedure.
   Program finds all numbers between 1 and Target that
   are divisible by the first six prime numbers.
*)
```

```
PROGRAM LEAVETest;

CONST Target = 15000;
VAR Count : INTEGER;

  PROCEDURE AwaitUser;
  VAR Ch: CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  Count := 0;
  WHILE Count <= Target DO
  BEGIN
    INC ( Count);
    (* as soon as a divisor is found, LEAVE to next value *)
    IF ( Count MOD 2) <> 0 THEN
      LEAVE
    ELSE IF ( Count MOD 3) <> 0 THEN
      LEAVE
    ELSE IF ( Count MOD 5) <> 0 THEN
      LEAVE
    ELSE IF ( Count MOD 7) <> 0 THEN
      LEAVE
    ELSE IF ( Count MOD 11) <> 0 THEN
      LEAVE
    ELSE (* if divisible by 2, 3, 5, 7, and 11 *)
      WRITELN ( Count);
  END; (* WHILE Count <= Target *)
  WRITELN ( 'Upon exit, Count = ', Count);
  AwaitUser;
END.
```

Compare the output from this program — shown in the following listing — with that from **CYCLETest**.

```
Upon exit, Count =        1
Press ENTER to continue
```

EXIT

The **EXIT** procedure ends the routine (procedure, function, or program) in which **EXIT** was called, transferring control to the block from which the terminating routines was called.

HALT

The **HALT** procedure ends the program immediately, transferring control back to the operating system. If you're experimenting with such commands, you probably don't need this book.

Summary

Congratulations, you've just mastered all of Pascal's control constructs! You'll be using these constructs extensively throughout the book and in the programs that you write on your own.

In this chapter, you learned about selection and iteration constructs. The selection constructs are: **IF**, **IF-ELSE**, extended **IF-ELSE**, and the **CASE** statement. The iteration constructs are the **FOR**, **REPEAT** and **WHILE** loops.

In the next chapter, you'll start putting this knowledge to use, as you learn how to define more complex procedures and functions.

6

Procedures & Functions Revisited

In Chapter 4, you learned about the general structure of procedures and functions (routines). You learned how to define routines, and how to specify and use parameters with these routines. In this chapter, we'll discuss various miscellaneous but important details about procedures and functions. You'll learn how to define procedures and functions inside of other routines. You'll also learn rules for determining which routines can call which other ones. Finally, you'll find an introduction to recursion, a very powerful programming technique in which a routine calls itself in order to help accomplish its task.

Calling Procedures and Functions

When the compiler processes your source file, it essentially processes the source code from start to finish. The compiler evaluates each statement to make sure it conforms to Pascal's syntax. Among other things, the compiler evaluates procedure and function calls.

Recall, from Chapter 4, that the argument list in a procedure or function call must correspond to the parameter list in the called routine's interface. In order for the compiler to evaluate such a call, it must know what the called routine's interface is.

Suppose procedures **A** and **B** are both defined in the main program. If procedure **A** calls **B**, then the compiler needs to know **B**'s interface to evaluate the syntactic correctness of the call. In practice, this would mean that procedure **B** must be defined before **A**.

But suppose procedure **B** calls **A** as well? For this to work, **A** must be defined before **B**. Is there any way around these contradictory requirements?

Pascal allows you to use a **FORWARD** declaration to specify just the interface for a procedure or function, while actually defining the routine later on. To do this, simply specify the procedure heading, followed by a semicolon. After the heading, put the reserved word **FORWARD**, followed by a semicolon. This tells the compiler not to expect the rest of the routine (that is, declarations and body). For example, the following line illustrates such a **FORWARD** declaration.

```
PROCEDURE CheckValues ( Val1, Val2 : REAL); FORWARD;
```

Later in the declaration section, you must include the actual definition for this procedure. The definition will include a parameterless heading, the routine's declaration section, and the procedure body.

For example, the following listing represents the definition for procedure **CheckValues**, after the **FORWARD** declaration in the previous listing:

```
(* no parameters, since they were listed in the FORWARD line
*)
PROCEDURE CheckValues;

  (* Declarations go here, as always *)

BEGIN
  (* procedure body here, as always *)
END;
```

The parameter list is not specified because the compiler already has that information. You may find it useful, however, to include the parameter information (within comments), as in either of the following alternative to the heading in the definition for **CheckValues**:

```
PROCEDURE CheckValues; (* Val1, Val2 : REAL *)
PROCEDURE CheckValues (* Val1, Val2 : REAL *);
```

TIP: In a FORWARD declaration, the procedure or function heading (with parameter list) is followed by the reserved word FORWARD. A semicolon must separate the procedure or function heading and the FORWARD, and a semicolon must separate the FORWARD from subsequent statements. The routine's declarations and body cannot be included with

the FORWARD declaration. The parameters cannot be included with the actual procedure or function definition.

Nested Routines

In Chapter 4, you learned that procedures and functions had the same structure as a program: heading, declarations, and body. Just as in the main program, you can define procedures and functions in a routine's declaration section. A routine that is defined inside another is said to be *nested* in the outer routine.

The following program provides examples of nested routines. Trace through the program to see the sequence in which routines are called. Look at the routine call-chain at various points in the program --- to see what routines have been started. (You can use the CodeWarrior Pascal debugger to do this. You'll find the call-chain in the stack crawl pane of the program window. See the *CodeWarrior User's Gu*ide for help on how to do this.)

Procedure **GetBoundedInt** uses two nested routines to get the desired value. Procedure **AdjustVals** makes any necessary adjustments so that the lower bound is less than the upper bound. This deals with cases where the **GetBoundedInt** procedure might be called with the middle two parameters reversed. Similarly, function **IsBetween** checks whether the value entered by the user falls within the specified bounds. If this function returns **TRUE**, the **REPEAT** loop can be terminated.

Listing 6.1

```
(* Program to show definition and use of nested routines *)

PROGRAM NestedExamples;

VAR Value : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Get an integer lying within the boundaries
     specified by MinVal and MaxVal
  *)
  PROCEDURE GetBoundedInt ( Message : STRING;
                            MinVal, MaxVal : INTEGER;
                            VAR Value : INTEGER);

    (* Fix boundary values if they are reversed ---
       that is, if Small > Large.
    *)
    PROCEDURE AdjustVals ( VAR Small, Large : INTEGER);
    VAR Temp : INTEGER;
    BEGIN
      WRITELN ( 'Start AdjustVals');
      IF Small > Large THEN
      BEGIN
        Temp := Small;
        Small := Large;
        Large := Temp;
      END;
      WRITELN ( 'Finish AdjustVals');
    END; (* AdjustVals *)

    (* Return TRUE if Small <= Val <= Large *)
    FUNCTION IsBetween ( Val, Small, Large : INTEGER) : BOOLEAN;
    BEGIN
      WRITELN ( 'Start IsBetween');
      IsBetween := ( Small <= Val) AND ( Large >= Val);
      WRITELN ( 'Finish IsBetween');
    END;  (* FN IsBetween *)

  BEGIN
    WRITELN ( 'Start GetBoundedInt');
    AdjustVals ( MinVal, MaxVal);
    REPEAT
      WRITE ( Message, ' ');
      READLN ( Value);
    UNTIL IsBetween ( Value, MinVal, MaxVal);
    WRITELN ( 'Finish GetBoundedInt');
  END; (* GetBoundedInt *)
```

```
BEGIN
  WRITELN ( 'Start main program');
  GetBoundedInt ( 'Value between -5 and 39:',
                  -5, 39, Value);
  WRITELN ( Value);
  AwaitUser;
END.
```

Procedure **AdjustVals** and function **IsBetween** are both nested in **GetBoundedInt**. This nesting routine provides the *immediate environment* for the two nested routines. Conversely, the two routines are said to be defined in **GetBoundedInt**'s *local environment.*

Such nested routines are accessible only while the nesting routine is executing. This restriction means that the main program cannot call either **AdjustVals** or **IsBetween**. (In fact, the main program doesn't even know that these routines exist.) This invisibility is a very useful property, since it makes it easier for you to isolate independent portions of your programs.

By limiting the number of routines that have access to a procedure or function, you can minimize the likelihood of bugs or errors resulting from improper use of that routine. Similarly, by limiting the contexts in which that procedure or function can be called, you can make it easier to identify the source of any errors involving the routine.

Using nested procedures also makes your source code easier to read. In the *parent* (that is, nesting) routine, you'll have a single procedure or function call, instead of the code that carries out the work of the nested routine. This advantage will be particularly useful later, when you need to write more complex procedures and functions.

This nesting process can be continued. That is, a nested routine can include nested routines of its own, and so forth. Later in the chapter, you'll see examples of multiple levels of nesting.

Program Design Issues

The ability to nest routines helps you to create procedures and functions that are independent of each other. It also enables you to hide the messy details of a task from the main program.

For example, suppose you're designing interactive software, and you'd like to make your programs as robust as possible. One task that's important for such a goal is to handle input properly. If your program is expecting numerical input and the user accidentally types a string, an ordinary Pascal program will crash.

To avoid this, you need to include a routine that reads information as a string and then converts the string to a numerical value of the appropriate sort. Such a routine has to check for various types of values and also for invalid input.

You can create a self-contained routine for doing this, by nesting the subtasks within the routine. Let's see how this is done, to illustrate some considerations when writing CodeWarrior Pascal programs. To keep things manageable, let's build an **INTEGER**.

The basic strategy will be to process the string, character by character, and to build a numerical value from these characters. The procedure stops when it finds an invalid character or reaches the end of the string. This procedure will perform the same sequence of actions repeatedly:

- Check whether the next character is a digit.
- If so, determine the numerical value of the digit; otherwise exit from the loop.
- If a digit was read, multiply the number so far by 10, and then add the digit's value to this running total.
- Update counters.

The procedure will also need to check a special case: whether the first character is a minus (-) or plus (+) sign.

The following listing includes a procedure, **MakeInt**, that implements such a strategy. Run the program, using valid and invalid values. For example, try the program with values such as 5, -35, 58g, "hello."

The procedure contains three nested routines to carry out some of the tasks described. To see how the procedure does its work, add some **WRITELN** statements that tell you where the program is at various points in its execution.

Listing 6.2

```
(* Program to illustrate procedure nesting and
   to provide MakeInt --- a more robust input
   handler for integers.
*)

PROGRAM IntDemo;
```

```
VAR  StrVal : STRING;
     Result :INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Build an integer value from a string.
     The first character can be a + or -; after that, only
     digits are allowed. The procedure stops processing as
     soon as a non-digit is encountered.
  *)
  PROCEDURE MakeInt ( Str : STRING;
                      VAR Result : INTEGER);
  CONST ErrorVal = 0;
        Base = 10;
  VAR Index, HowLong : INTEGER;
      Negative : BOOLEAN;

    (* Returns TRUE if Ch is a digit character. *)
    FUNCTION IsDigit ( Ch : CHAR) : BOOLEAN;
    BEGIN
      IsDigit := ( Ch >= '0') AND  ( Ch <= '9');
    END; (* IsDigit *)

    (* Returns the numerical value of the 'digit" Ch *)
    FUNCTION ToVal ( Ch : CHAR) : INTEGER;
    BEGIN
      ToVal := ORD ( Ch) - ORD ( '0');
    END; (* ToVal *)

    (* Set Negative if the number starts with a minus
       sign; skip the first character if it's a +
    *)
    PROCEDURE HandleFirstCh;
    BEGIN
      IF Str [ 1] = '-' THEN
      BEGIN
        Negative := TRUE;
        INC ( Index);
      END
      ELSE IF Str [ 1] = '+' THEN
        INC ( Index);
    END; (* HandleFirstCh *)

  BEGIN  (* MakeInt *)
    Result := 0;
    Negative := False;
    HowLong := LENGTH ( Str);
    IF HowLong > 0 THEN
    BEGIN
      Index := 1;
      HandleFirstCh;
      WHILE ( Index <= HowLong) DO
      BEGIN
        If IsDigit ( Str [ Index]) THEN
        BEGIN
          Result := Base * Result + ToVal ( Str [ Index]);
          INC ( Index);
        END
        ELSE
          Index := HowLong + 1;
      END;  (* WHILE HowLong > 0 *)
      IF Negative THEN
        Result := Result * (-1);
    END
    ELSE  (* If a null string was passed, return ErrorVal *)
      Result := ErrorVal;
  END; (* MakeInt *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

BEGIN
  REPEAT
    GetString ( 'Value? (> 500 to stop)', StrVal);
    MakeInt ( StrVal, Result);
    WRITELN ( StrVal, ' : ', Result);
  UNTIL Result > 500;
  AwaitUser;
END.
```

By moving the code for these actions to nested procedures, you can simplify the code for the actual **MakeInteger** routine. A single-line call to **HandleFirstCh**, for example, is much easier to read and understand than the source code for the procedure. However, there is a price to pay for this improvement in code clarity: the routine executes a bit more slowly, since it takes time to invoke a procedure or function.

By defining the utility functions **ToVal** and **IsDigit** within **MakeInt**, you make the procedure independent of anything but predefined routines (such as **ORD**). If the procedure doesn't work correctly, you know that the error must lie somewhere within the definition for **MakeInt**.

Reading Source Files During Compilation

The **MakeInt** routine is a more robust way of getting whole number input from the user. Unlike **GetInteger**, this routine won't crash or evaluate to zero if the user accidentally types a letter instead of a number. This is a good routine to use in programs that need to get integer input.

So far, you've been copying **GetInteger** to each new source file, or perhaps even typing it in from scratch. This would become quite tedious with **MakeInt**, which is much longer than **GetInteger**.

Fortunately, there's a way of handling such a situation. You can keep the source code for **MakeInt** (and perhaps also for a modified version of **GetInteger**) in a separate file. If you want to use these routines in a program, you can tell the compiler to read the contents of the file at the appropriate point in the main program's declaration section.

To do this, you need to use a *compiler directive* --- an instruction to the compiler to set certain flags or to carry out a particular action. For example, suppose you have the contents of **MakeInt** and **GetInteger** in a file named **IO.PAS**.

The following listing shows the contents of this file. Notice the modification in **GetInteger** --- to get a numerical value by calling **MakeInt**.

```
(* IO.PAS:
   Contains MakeInt and GetInteger.
   NOTE: This file is not a program; it does not compile.
*)

  (* Build an integer value from a string.
     The first character can be a + or -; after that, only
     digits are allowed. The procedure stops processing as
     soon as a non-digit is encountered.
  *)
  PROCEDURE MakeInt ( Str : STRING;
                      VAR Result : INTEGER);
  CONST ErrorVal = 0;
        Base = 10;
  VAR Index, HowLong : INTEGER;
      Negative : BOOLEAN;

    (* Returns TRUE if Ch is a digit character. *)
    FUNCTION IsDigit ( Ch : CHAR) : BOOLEAN;
    BEGIN
      IsDigit := ( Ch >= '0') AND  ( Ch <= '9');
    END; (* IsDigit *)

    (* Returns the numerical value of the 'digit" Ch *)
    FUNCTION ToVal ( Ch : CHAR) : INTEGER;
    BEGIN
      ToVal := ORD ( Ch) - ORD ( '0');
    END; (* ToVal *)

    (* Set Negative if the number starts with a minus
       sign; skip the first character if it's a +
    *)
    PROCEDURE HandleFirstCh;
    BEGIN
      IF Str [ 1] = '-' THEN
      BEGIN
        Negative := TRUE;
        INC ( Index);
      END
      ELSE IF Str [ 1] = '+' THEN
        INC ( Index);
    END; (* HandleFirstCh *)

  BEGIN  (* MakeInt *)
    Result := 0;
  Negative := False;
  HowLong := LENGTH ( Str);
  IF HowLong > 0 THEN
  BEGIN
    Index := 1;
    HandleFirstCh;
    WHILE ( Index <= HowLong) DO
    BEGIN
      If IsDigit ( Str [ Index]) THEN
      BEGIN
        Result := Base * Result + ToVal ( Str [ Index]);
        INC ( Index);
      END
      ELSE
        Index := HowLong + 1;
    END;  (* WHILE HowLong > 0 *)
    IF Negative THEN
      Result := Result * -1;
  END
  ELSE  (* If a null string was passed, return ErrorVal *)
    Result := ErrorVal;
END; (* MakeInt *)

(* Prompt user and read an INTEGER.
   Modified to call MakeInt.
*)
PROCEDURE GetInteger ( Message : STRING;
                       VAR Value : INTEGER);
VAR TempStr : STRING;
BEGIN
  WRITE ( Message, ' ');
  READLN ( TempStr);
  MakeInt ( TempStr, Value);
END; (* GetInteger *)
```

Once you have created this file, you can include its contents in other files. The following program shows how to include a file in another one.

Listing 6.3

```
(* Program to illustrate use of INCLUDE $I directive. *)

PROGRAM IncludeDemo;
```

```
VAR IntVal : INTEGER;

(* The following line tells compiler to read IO.PAS
   in the specified folder; you'll want to replace the path
   to match your configuration.
*)
(*$I h:\CWPASC\CH6\IO.PAS *)

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  GetInteger ( 'Value?', IntVal);
  WRITELN ( 'You entered ', IntVal);
  AwaitUser;
END.
```

The compiler directive --- the line beginning with **(*$I** --- tells the compiler to read the contents of the specified file (**IO.PAS**) at that point in the file. The effect is the same as if the source text from that file were physically included in the program source file. The compiler includes the source code, and compiles it, just as all other text in the program is compiled.

The **$I** must follow the comment character(s) immediately --- otherwise the directive will be interpreted as a comment rather than as a compiler directive.

Accessibility of Routines and Variables

Since routines can be nested, the rules that govern what can call what can get a bit involved. The same identifiers may be used in different places in a program, which further complicates the issue. This section and the next discuss CodeWarrior Pascal's accessibility and naming rules.

Program **Shell1** is shown in the next listing and is extended in subsequent listings. You may want to fill in actual statements --- to test the assertions in this section with specific examples.

The accessibility rules are marked with bullets in the text.

Listing 6.4

```
(* Program shell to illustrate procedure and function
   nesting and accessibility.
   This version contains just two level 1 routines.
*)

PROGRAM Shell1;
  PROCEDURE G1; FORWARD;   (* So A1 can call G1 *)

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE ReportLocation ( Message, Location : STRING);
  BEGIN
    WRITELN ( Message, ‘ from ‘, Location);
  END; (* Report Location *)

  PROCEDURE A1;
  BEGIN (* A1 *)
    ReportLocation ( ‘Hello  ’, ‘A1’);
    ReportLocation ( ‘Goodbye’, ‘A1’);
  END; (* A1 *)

  PROCEDURE G1;
  BEGIN (* G1 *)
    ReportLocation ( ‘Hello  ’, ‘G1’);
    A1;
    ReportLocation ( ‘Goodbye’, ‘G1’);
  END; (* G1 *)

BEGIN (* Main program *)
  ReportLocation ( ‘Hello  ’, ‘Main’);
  G1;
  WRITELN ( 'Done with Shell1 program.');
  AwaitUser;
  ReportLocation ( ‘Goodbye’, ‘Main’);
END.
```

The preceding program provides a shell for procedure calls. In its current form, **Shell1** contains two procedures (**A1** and **G1**), a couple of utility routines (**AwaitUser** and **ReportLocation**), and the main program. The two procedures don’t really do anything except display information about their comings and goings.

The following discussion introduces routines defined at different “levels" --- that is, nested to varying depths. For purposes of the discussion, think of the main program as level 0. Routines defined in the declaration section of the main program are at level 1.

- In Pascal, the main program can call any procedure or function defined in the program's declaration section --- that is, it can call any level 1 routines. The main program is the *immediate environment* for these routines. For example, procedures **A1** and **G1** are level 1 procedures in the preceding program.
- Level 1 routines can call each other --- provided that the interface information is available, as discussed earlier in the chapter. Thus, the **FORWARD** declaration for **G1** is necessary if **A1** is to call **G1**. (Notice the format for a parameterless **FORWARD** declaration.)

A level 1 routine can have level 2 procedures and functions defined in its local environment. For example, program **Shell2**, in the following listing, contains three level 2 routines: **B2** and **D2** are nested in **A1**, and **H2** is nested in **G1**. H2 is defined in **G1**'s local environment; conversely, **G1** is the immediate environment for **H2**. Notice that **D2**'s **FORWARD** declaration is made in **D2**'s immediate environment (that is, in the declaration section of **A1**), not in the main program.

As an exercise, you might want to copy ReportLocation to the Shell2 program. Then test your knowledge of calling sequences by trying to predict all the routines that each routine in the program can call.)

Listing 6.5

```
(* Program shell to illustrate procedure and function
   nesting and accessibility.
   This version contains level 1 and level 2 routines.
*)

PROGRAM Shell2;
  PROCEDURE G1; FORWARD;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE A1;             (* Level 1 *)
    (* Note that this FORWARD declaration is in
       A1 not in the main program --- because
       D2 is defined inside A1, and not in the main program.
    *)
    PROCEDURE D2; FORWARD;

    PROCEDURE B2;           (* Level 2 *)
    BEGIN  (* B2 *)
    END; (* B2 *)

    PROCEDURE D2;           (* Level 2 *)
    BEGIN  (* D2 *)
    END; (* D2 *)

  BEGIN (* A1 *)
  END; (* A1 *)
  PROCEDURE G1;             (* Level 1 *)

    PROCEDURE H2;
    BEGIN  (* H2 *)
    END; (* H2 *)

  BEGIN (* G1 *)
  END; (* G1 *)

BEGIN (* Main program *)
  WRITELN ( 'Done with Shell2 program.');
  AwaitUser;
END.
```

- A level 1 routine can call any level 2 routines nested within it, but cannot call any other level 2 routines. Thus, **A1** can call both **B2** and **D2**, but cannot call **H2**. Similarly, **H2** is the only level 2 routine that **G1** can call.
- The level 2 routines *within a procedure or function* can call each other --- again, provided the necessary interface information is available. Thus, **B2** and **D2** can call each other --- because **D2** has been **FORWARD** declared. On the other hand, level 2 routines nested within different routines cannot call each other. Thus, **H2** cannot call either **B2** or **D2**.
- Level 2 routines can call any level 1 routine --- provided the appropriate interface information is available to level 2's immediate environment. For example, **H2** can call **A1**; D2 can call **G1** because the latter procedure was **FORWARD** declared.
- The main program *cannot* call any level 2 routines.

Level 2 routines can also have nested routines, which would be at level 3. In program **Shell3**, C3 and **E3** are level 3 routines.

Listing 6.6

```
(* Program shell to illustrate procedure and function
   nesting and accessibility.
   This version contains routines from levels 1, 2, and 3.
*)

PROGRAM Shell3;
  PROCEDURE G1; FORWARD;

  PROCEDURE A1;                (* Level 1 *)
    PROCEDURE D2; FORWARD;

    PROCEDURE B2;              (* Level 2 *)

      PROCEDURE C3;            (* Level 3 *)
      BEGIN (* C3 *)
      END; (* C3 *)

    BEGIN  (* B2 *)
    END; (* B2 *)

    PROCEDURE D2;               (* Level 2 *)

      PROCEDURE E3;             (* Level 3 *)
      BEGIN (* E3 *)
      END; (* E3 *)

    BEGIN  (* D2 *)
    END; (* D2 *)

  BEGIN (* A1 *)
  END; (* A1 *)

  PROCEDURE G1;                 (* Level 1 *)

    PROCEDURE H2;               (* Level 2 *)
    BEGIN  (* H2 *)
    END; (* H2 *)

  BEGIN (* G1 *)
  END; (* G1 *)

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN (* Main program *)
  WRITELN ( 'Done with Shell3 program.');
  AwaitUser;
END.
```

- Level 3 routines can call any of the following: level 1 routines with a known interface, the (level 2) routine that represents their immediate environment and any level 2 routines their immediate environment can call. Thus, **C3** can call **A1** and **G1** because they are both level 1; **C3** can call **B2** because **B2** is the immediate environment for **C3**; and **C3** can call **D2** because this is a level 2 routine that **B2** can call. An interface is known to a level 3 routine if the interface is accessible to the routine's immediate environment.
- Level 3 routines defined within the same routine can call each other,
-
- but they *cannot* call level 3 routines nested inside other routines. Thus, **C3** and **E3** cannot call each other.
- Level 1 routines cannot call any level 3 routines.
- A level 2 routine can call only the level 3 routines nested within it, but no other level 3 routines. Thus, **B2** can call **C3** but not **E3**. On the other hand, **D2** can call **E3** but not **C3**.

Such nesting can continue to arbitrary levels. For example, program **Shell4** (in the next listing) shows a level 4 procedure, **F4**, nested within **E3**.

Listing 6.7

```
(* Program shell to illustrate procedure and function
   nesting and accessibility.
   This version contains routines from levels 1, 2, 3, and 4.
*)

PROGRAM Shell4;
  PROCEDURE G1; FORWARD;

  PROCEDURE A1;                (* Level 1 *)
    PROCEDURE D2; FORWARD;

    PROCEDURE B2;              (* Level 2 *)
```

```
      PROCEDURE C3;             (* Level 3 *)
      BEGIN (* C3 *)
      END; (* C3 *)

    BEGIN  (* B2 *)
    END; (* B2 *)

    PROCEDURE D2;               (* Level 2 *)

      PROCEDURE E3;             (* Level 3 *)

        PROCEDURE F4;           (* Level 4 *)
        BEGIN (* F4 *)
        END; (* F4 *)

      BEGIN (* E3 *)
      END; (* E3 *)

    BEGIN  (* D2 *)
    END; (* D2 *)

  BEGIN (* A1 *)
  END; (* A1 *)

  PROCEDURE G1;                 (* Level 1 *)

    PROCEDURE H2;               (* Level 2 *)
    BEGIN  (* H2 *)
    END; (* H2 *)

  BEGIN (* G1 *)
  END; (* G1 *)

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)
```

```
BEGIN (* Main program *)
  WRITELN ( 'Done with Shell4 program.');
  AwaitUser;
END.
```

With this level you can finally establish some general rules for accessibility. All statements assume that any necessary interface information is available --- either through **FORWARD** declarations or through the relative position of the routines.

- A level 4 routine can access any level 5 routines defined in its local environment.
- A level 4 routine can access any level 4 routines nested within the same immediate environment.
- A level 4 routine can access its nesting routine --- that is, the routine that constitutes the routine's immediate environment. Thus, **F4** can access **E3**.
- A level 4 routine can access any level 3 and level 2 routines accessible to its nesting routine. Thus, **F4** could access any level 3 routines that would be defined in **D2**, as well as any level 2 routines accessible to **E3**. In this case, the latter condition would make **B2** and **D2** accessible.
- Like routines at all levels, a level 4 routine can access any level 1 routines.

The following list shows what is accessible to routines at various levels. The list assumes that any necessary **FORWARD** declarations have been made. Also, keep in mind that a routine can, in principle, call itself.

- Main Program : All level 1 routines.
- Level 1 Routines : All level 1 routines with known interface + Any level 2 routines defined in the local environment
- Level 2 Routines : All level 1 routines with known interface + Any level 2 routines defined in the same immediate environment + Any level 3 routines defined in local environment
- Level 3 Routines : All level 1 routines with known interface + Any level 3 routines

defined in the same immediate environment + The nesting routine + Any level 2 routines accessible to the nesting routine + Any level 4 routines defined in local environment

- Level *n* Routines : All level 1 routines with known interface + Any level *n* routines defined in the same immediate environment + The nesting routine + Any level *n-1*, n-2, ..., 3, 2 routines accessible to the nesting routine + Any level *n+1* routines defined in local environment

No procedure can call any routines more than one level deeper than itself. For example, a level 6 routine could never call a level 8 routine. The routine could, however, call any level 7 routines defined in its local environment.

Scope and Accessibility

As you've seen, you can't just call a procedure or function from anywhere in your program. The routine is accessible only in certain contexts. The contexts within which a routine is accessible represent the routine's *scope*. A procedure or function is said to be *within scope* if that routine is accessible at a given point in a program.

The summary of accessibility in the previous section shows that level 1 routines are always within scope. This is why such routines are known as *global routines*. All other routines have more limited scope, and are generally known as *local routines*.

The concept of scope also applies to constants, types and variables. That is, there are rules concerning the accessibility of variables and other declarations in a program. The rules for constants and types are the same as for variables.

The sample program shell from **Shell4** shows these rules. Suppose there is a reference to a variable in procedure **F4**. How does the program search for the memory that corresponds to the identifier?

First, the routine's local environment is checked --- in this case, the declaration section of **F4**. If the variable is not found there, the declaration section of the immediate environment (**E3**, in the example) is checked.

If the variable is still not found then the declaration section of the immediate environment's immediate environment (**D2**, in the example) is searched. This process continues through successive immediate environments, until the variable is found or until the main program is reached. The main program (the global environment) is the last place searched.

TIP: The local environment of the currently executed routine is checked first whenever the program needs to find

the storage associated with a variable name or a procedure or function body.

If a routine can search a particular environment, the routine can “see" declarations and definitions in that environment. Thus, **F4** can see its local environment, as well as the declaration sections of **E3**, D2, **A1**, and the main program. **F4** cannot see **B2**, G1, and so forth.

Note that there is a difference between being able to call a routine and being able to see that routine's local environment. For example, **F4** cannot see either **B2** or **G1**, but can call both of these routines.

In general, a routine may be able to call another one but will not necessarily be able to see it. For example, **E3** can call **F4**, but cannot see it. On the other hand, if a routine can see another one, then the routine can also call it. Thus, **F4** can both call and see **E3**.

A variable is within scope of any routine that can see the environment within which the variable is declared. Thus, a variable declared in **A1** is within scope in **A1**, B2, **D2**, C3, **E3**, and **F4**, but *not* in **G1** or **H2**.

Visibility

You can use the same identifiers in more than one place in your programs. For example, the following program uses the same identifier (**Val**) in three different places. Three different memory locations are given the same name.

Listing 6.8

```
(* Program to illustrate visibility of names *)

PROGRAM Visibility;

VAR Val : INTEGER;    (* Global declaration *)

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)
  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

  PROCEDURE Change ( VAR Value : INTEGER);
  VAR Val : INTEGER;    (* Local declaration *)

    FUNCTION OKToChange ( Val : INTEGER) : BOOLEAN;
    BEGIN
      OKToChange := Val <= ( MAXINT DIV 2);
    END; (* FN OKToChange *)

  BEGIN
    GetInteger ( 'Value? (0 to stop)', Val);
    IF OKToChange ( Val) THEN
      Value := Val * 2
    ELSE
      Value := Val DIV 2;
    WRITELN ( 'In Change, Val = ', Val);
  END; (* Change *)

BEGIN (* Main program *)
  REPEAT
    Change ( Val);
    WRITELN ( 'In main, Val = ', Val);
  UNTIL Val = 0;
  AwaitUser;
END.
```

In the example program, the global **Val** is within scope for all contexts --- the main program, **Change** and **OKToChange**. The local **Val**, declared within **Change**, is within scope only for that procedure and for **OKToChange**. Finally, **OKToChange**'s parameter **Val** is within scope only for the function.

What is referenced when **Val** is used in a particular location depends on the identifier's visibility. An identifier's *visibility* determines the contexts in which the identifier refers to a specific *object* (that is, a memory location or a routine). If an object is out of scope, the object has no visibility. If the object is within scope in a particular context, that object may or may not be visible.

In a particular context, if two objects have the same name, the object with the greater visibility will be referenced if the identifier is used in a statement. For example, within function **OKToChange**, the formal parameter has a greater visibility than either the **Val** declared in **Change** or the global **Val** --- even though all three are within scope.

An analogy might clarify the distinction between scope and visibility. Suppose you hang around with a group of people who are somewhat familiar with programming and programming languages. For example, assume they have heard of Pascal and of Niklaus Wirth.

Now, suppose one of your group of friends is named Emma Wirth. Since both Niklaus and Emma Wirth are known to your group, both of these “objects" have scope among your friends.

What if someone asks “What's Wirth working on these days?" at a get-together with your friends. The assumption would probably be that the question is about Emma. In that context, Emma Wirth has greater visibility than Niklaus Wirth.

Suppose the same question were asked at the Eidgenossische Technische Hochschule in Zurich (where Niklaus Wirth teaches). Emma Wirth is probably unknown there, so that she would not be within scope. In that case, Niklaus Wirth would automatically have greater visibility than Emma Wirth.

To determine the relative visibility of two identical identifiers, you need to know how the program decides which object you mean when you use an identifier in the program.

The basic strategy used by the program is to search outward, beginning with the local environment. As soon as a match is found, the program stops searching. The match that is found has the greatest visibility of all the candidates within scope.

When a reference is made to a variable or procedure in a routine, the following search strategy is implemented and continues until an object or routine with the appropriate name is found.

- The local environment (which includes the routines's own interface) is checked first.
- The immediate environment is checked next.
- The immediate environment's immediate environment is checked.
- Successive immediate environments are checked until the main program's declaration section is reached. This is the last place checked.

This is why the formal parameter **Val** is referenced in **OKToChange**, whereas the local variable declared in **Change** is used when that procedure is executing. The global **Val** is used only in the main program, since the level 1 and level 2 routines each have the same identifier.

Note that the storage allocated for the global variable, **Val**, is accessible in procedure **Change**, despite the fact that the global identifier **Val** is *visible* only in the main program. In fact, this location is referenced by means of the variable parameter, **Value**. Changes to the **Value** parameter in the procedure are also changes to the global **Val**.

Recursion

Earlier you learned about the flow of control when a program is executing and routines are called. Generally, which a routine is temporarily suspended while another routine executes, and is then reactivated when the called routine finishes its work.

For example, if the main program calls procedure **First**, the program is suspended temporarily, and **First** begins executing. Suppose **First** calls **Second**. At that point, **First** is also suspended --- so there are now two suspended routines. If **Second** calls **Third** during execution, then **Second** will become another suspended routine --- number three.

If **Third** doesn't call any new routines, then it will eventually finish executing. When this happens, control will pass back to the caller --- in this case, **Second**. If **Second** also calls no new routines, then it will eventually finish; **First** will reactivate at that point. The last routine to be reactivated will be the main program, since it was the first to be suspended.

There is nothing strange about this process. The important thing to keep in mind is that procedures and functions are reactivated backwards --- the most recently suspended is the first to be reactivated, and so forth.

A special and powerful case of this process in action is a *recursive routine* --- a routine that calls itself. This means that one version of the routine (the caller) is suspended, and a new version (the called) is activated.

A recursive routine is given a task to do. If the task can be done in a single step, the routine will do it, and will return control to the caller. Otherwise, the procedure or function will pass a simpler form of the task to a new version of the routine. Thus, each caller passes an easier version of the task to the new routine.

Eventually, the problem will be solvable in a single step. That version of the routine will thus finish its work, and will return control to the caller. The temporarily suspended versions of the routine will generally do their part of the problem after the result has come back from each called routine.

Let's look at an example. In the following program, procedure **Count** is recursive. The version of **Count** used displays information about the call's position in the recursion.

Listing 6.9

```
(* Program to illustrate use of recursion in a procedure.
   Program counts to a specified number.
*)
PROGRAM RecurseDemo;

VAR Level, CountVal : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Prompt user and read an INTEGER. *)
  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  VAR TempStr : STRING;
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

   (* Recursive procedure to count to a specified value *)
  PROCEDURE Count ( Val : INTEGER);
  BEGIN  (* Count *)
    (* Display information about the procedure call *)
    Level := Level + 1;
    WRITELN ( ' ':Level * 2, 'Starting count, level = ',
              Level:2, '; val = ', Val);

    (* The actual Count routine is contained in the
       next three lines.
    *)
    IF ( Val > 1) THEN
      Count ( Val - 1);
    WRITELN ( ' ':Level * 2, Val);

    (* Display information about the procedure call *)
    WRITELN ( ' ':Level * 2, 'Leaving count, level  = ',
              Level:2, '; val = ', Val);
    Level := Level - 1;
  END;  (* Count *)

BEGIN
  REPEAT
    Level := 0;
    GetInteger ( 'Count to what value? ( 0 to stop)', CountVal);
    Count ( CountVal);
  UNTIL CountVal = 0;
  AwaitUser;
END.
```

The recursion in this program occurs in **Count**. When asked to count to six, this procedure produced the following output:

```
Count to what value? 6
  Starting count, Level =  1; Val = 6
    Starting count, Level =  2; Val = 5
      Starting count, Level =  3; Val = 4
        Starting count, Level =  4; Val = 3
          Starting count, Level =  5; Val = 2
            Starting count, Level =  6; Val = 1
            1
            Leaving count, Level  =  6; Val = 1
          2
          Leaving count, Level  =  5; Val = 2
        3
        Leaving count, Level  =  4; Val = 3
      4
      Leaving count, Level  =  3; Val = 4
    5
  Leaving count, Level  =  2; Val = 5
6
Leaving count, Level  =  1; Val = 6
```

The program displays the output in a manner that should make it easier to determine what happened at various levels of recursion. Thus, the six lines beginning with *Starting* are all displayed in calls to **Count**. Each of these calls is with a different argument, indicated by the number written after the introductory message.

Notice how each version of **Count** calls a copy of itself, but with a smaller argument (a simpler problem). This process stops when the program gets to **Count(1)**.

At this point, the currently executing version of **Count** can perform its task --- displaying the parameter's value --- right away. Thus, **Count(1)** actually displays its result.

In its current form, **Count(1)** also writes a “Leaving" message, to make it easier to follow the flow of control. All these elements are written at the same level of indentation. After this message is written, the value of **Level** is decreased by one, since the program is moving back up to the previous level of calls. **Level** is a variable that keeps track of the number of suspended procedures and functions.

The routine then returns control to its caller --- **Count(2)**. This routine executes the statement after the call to **Count(1)**. This statement tells **Count(2)** to display the current value of **Val**, which is 2 in that version of **Count**.

Notice that this call to **WRITELN** is *not* in an **ELSE** clause. This action is carried out by each version of **Count**. Thus, the successively higher values of **Val** are written as control reverts back to the appropriate versions of **Count**.

The Flow of a Recursive Call

Notice the relationship between the order in which the routines are completed and the order in which they are called. As you saw earlier, the last routine to be called is the *first* one to be finished. This also means that the first procedure or function to be suspended is the last one to finish.

You can see this in the program's indentation pattern. The program begins and ends at the leftmost lines. The lines indented furthest are those in the middle of the program, and these lines are consecutive.

Because the procedure's parameter is passed by value, each version of **Count** keeps its own copy of **Val**. This makes it possible for the different versions to have different values.

Features of a Recursive Routine

Before we look at another example, let's look at the unembellished form of the recursive procedure in the preceding program. The following listing, which is not a program, contains a clean version of **Count**. You can substitute this version for the annotated version in **RecurseDemo**.

```
(* Recursive procedure to count to a specified value *)
PROCEDURE Count ( Val : INTEGER);
BEGIN  (* Count *)
    IF ( Val > 1) THEN
      Count ( Val - 1);
    WRITELN ( Val);
END;  (* Count *)
```

Without the statements that write information about the recursion level, this procedure turns out to be very simple. Nevertheless, **Count** has all the features a recursive routine needs. In particular, a recursive routine must include the following types of statements:

1. The statements needed to carry out the task of the routine. In the **Count** procedure, this is the call to **WRITELN**.

2. A test to determine whether the routine can avoid calling itself again --- that is, to determine whether the routine can simply do its work and terminate. In the preceding procedure, this is the **IF** statement testing whether (**Val** > 1).

3. A statement that calls the routine itself. This call *must* be made with parameter values that will eventually make the test condition take on a value that will bypass the recursive call. In the preceding routine, this is accomplished by calling the procedure with arguments that approach the termination value of 1.

The routine must test whether a recursive call is necessary *before* making such a call. If you write your routine so that the routine calls itself first, then tests whether it's all right to stop, the routine will never end. Thus, the following version of the recursive procedure will never end:

```
(* FAULTY recursive counting procedure.
   !! DO NOT USE this version, since it will not work.
*)
PROCEDURE Count ( Val : INTEGER);
BEGIN  (* Count *)
```

```
    (* Because the recursive call is made first,
       the procedure never tests whether the call is
       necessary. Instead, the routine just keeps calling.
    *)
    Count ( Val - 1);
    IF ( Val > 1) THEN
      WRITELN ( Val);
END;  (* Count *)
```

Similarly, the following version of the procedure is incorrect because the value of **Val** used in each call is moving *away* from the termination value of 1. This version will also end in a run-time error.

```
(* FAULTY recursive counting procedure.
   !! DO NOT USE this version, since it will not work.
*)
PROCEDURE Count ( Val : INTEGER);
BEGIN  (* Count *)
    IF ( Val > 1) THEN
      (* Val INCREASES, so it will never be 1;
         therefore the routine will never terminate.
      *)
      Count ( Val + 1);
    WRITELN ( Val);
END;  (* Count *)
```

TIP: A recursive routine must always test whether it can stop *before* calling another version of itself. If a recursive call is made, the parameters in some call should eventually have values that will make further recursive calls unnecessary.

Another Example

Let's look at one more recursive routine --- in which there are *two* recursive calls within the same routine. This procedure generates *Fibonacci numbers*, which show up in all sorts of contexts.

Fibonacci numbers are named after Leonardo of Pisa (also known as Fibonacci). He discovered these numbers as the solution to the following problem about the rate at which rabbits reproduce. How many pairs of rabbits will be produced in one year, given the following conditions:

- You begin with a single pair
- Every month each pair begets a new pair
- A rabbit pair becomes productive from the second month on

At the start, and after the first month, you have just the one pair of rabbits. Thus, the first two Fibonacci numbers --- **Fib(1)** and **Fib(2)** --- are 1.

Subsequent Fibonacci numbers are formed by adding the *two* preceding Fibonacci numbers. Thus, in the third month, you will have two pairs: the original pair and the pair produced by them. In the fourth month, you will have these two pairs and another pair produced by the original rabbits. Thus, **Fib(3)** is 2 (1 + 1), and **Fib(4)** is 3 (1 + 2).

In the fifth month, you'll have the three pairs, plus two new pairs: one from the original pair and one pair from the oldest pair of offspring: **Fib(5)** is 5 (3 + 2).

Fibonacci numbers show up in many other contexts. For example, the growth of leaves on a plant stem follows a Fibonacci sequence in many plants. To see this, locate three leaves on a plant that are directly above each other on the stem. Count the number of leaves on the stem between the lowest and the middle of the three leaves. Then count the number between the middle and the highest leaf. These two numbers will generally be successive terms in the Fibonacci sequence. (This, of course, assumes that leaves haven't fallen off or been removed for other reasons.)

Mathematically, a Fibonacci number is defined recursively --- that is, in terms of itself. the definition is as follows:

The first two Fibonacci numbers --- **Fib(1)** and **Fib(2)** --- are defined to be 1.

- Each successive Fibonacci number is defined as the sum of the two preceding Fibonacci numbers. That is,

Fib(x) = Fib(x - 1) + Fib(x - 2)

Table 7: The First 15 Fibonacci Numbers

Number	Value	Source
Fib(1)	1	Definition
Fib(2)	1	Definition
Fib(3)	2	Fib(1) + Fib(2)
Fib(4)	3	Fib(2) + Fib(3)
Fib(5)	5	Fib(3) + Fib(4)
Fib(6)	8	Fib(4) + Fib(5)
Fib(7)	13	Fib(5) + Fib(6)
Fib(8)	21	Fib(6) + Fib(7)
Fib(9)	34	Fib(7) + Fib(8)
Fib(10)	55	Fib(8) + Fib(9)
Fib(11)	89	Fib(9) + Fib(10)
Fib(12)	144	Fib(10) + Fib(11)
Fib(13)	233	Fib(11) + Fib(12)
Fib(14)	377	Fib(12) + Fib(13)
Fib(15)	610	Fib(13) + Fib(14)

The following program contains a recursive procedure to generate Fibonacci numbers.

Listing 6.10

```
(* Program to illustrate recursion using
   Fibonacci number generation.
*)
```

```
PROGRAM FibonacciDemo;

VAR FibVal : INTEGER;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Prompt user and read an INTEGER. *)
  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  VAR TempStr : STRING;
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

   (* Compute the specified Fibonacci number.
      NOTE: Don't call this procedure for values larger than
            about 25 or 30; if you do, you'll be waiting a
            l o n g time for the routine to finish.
  *)
  FUNCTION Fib ( Val : INTEGER) : LONGINT;
  BEGIN
    IF ( Val <= 2) THEN
      Fib := 1
    ELSE
       (* Two other Fibonacci numbers must be computed *)
      Fib := Fib ( Val - 2) + Fib ( Val - 1);
  END;  (* FN Fib *)

BEGIN
  REPEAT
    GetInteger ( 'Compute what Fibonacci value? (1 to stop)',
                 FibVal);
    WRITELN ( 'Fib ( ', FibVal, ') = ', Fib ( FibVal));
  UNTIL FibVal = 1;
  AwaitUser;
END.
```

Procedure **Fib** computes the desired Fibonacci number using recursion. Let's see how this routine does its work.

If the argument is small --- 1 or 2 --- the procedure returns a value immediately; otherwise, the routine ships the problem out to *two* different versions of itself. One version computes the preceding Fibonacci number (**Fib(n - 1)**), and the other version computes the Fibonacci number before that (**Fib(n - 2)**). Each of these calls may, in turn, call other versions of **Fib**.

As you might guess, the calls to **Fib** quickly proliferate. This fact raises an important point about recursion: Although it can be a very powerful technique in some situations, recursion can also be very inefficient.

The following program shows how quickly the number of calls to **Fib** increases, as you ask for larger Fibonacci numbers.

Listing 6.11

```
(* Program to compute a specified Fibonacci number,
   and also to keep track of the number of recursive calls.
   Program also illustrates recursion.
*)
PROGRAM LoudFibonacciDemo;

VAR FibVal : INTEGER;
    Calls : LONGINT;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Prompt user and read an INTEGER. *)
  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  VAR TempStr : STRING;
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

  FUNCTION LoudFib ( Val : INTEGER) : LONGINT;
  BEGIN
    Calls := Calls + 1;
    IF ( Val <= 2) THEN
      LoudFib := 1
    ELSE
      LoudFib := LoudFib ( Val - 2) + LoudFib ( Val - 1);
  END;  (* FN LoudFib *)

BEGIN
  REPEAT
    Calls := 0;
    GetInteger ( 'Compute what Fibonacci value? (1 to stop)',
                 FibVal);
    WRITE ( 'Fib ( ', FibVal:2, ') = ', LoudFib ( FibVal):10);
    WRITELN ( ';  ', Calls:10, ' calls to Fib');
  UNTIL FibVal = 1;
  AwaitUser;
END.
```

This program produces the following results for a range of values:

```
Fib (  5) =          5;           9 calls to Fib
Fib ( 10) =         55;         109 calls to Fib
Fib ( 15) =       610;       1219 calls to Fib
Fib ( 20) =      6765;      13529 calls to Fib
Fib ( 25) =     75025;     150049 calls to Fib
Fib ( 26) =    121393;     242785 calls to Fib
Fib ( 27) =    196418;     392835 calls to Fib
Fib ( 28) =    317811;     635621 calls to Fib
Fib ( 29) =    514229;    1028457 calls to Fib
Fib ( 30) =    832040;    1664079 calls to Fib
```

Notice that the number of calls gets very high, even for relatively small values of **FibVal**.

Iterative Solutions to Recursive Problems

Sometimes a non-recursive, or *iterative*, solution to a problem is much quicker. You can always find an iterative way to do a task. For example, the following listing contains an iterative Fibonacci function.

Add this function definition to the **FibonacciDemo** or the **LoudFibonacciDemo** programs. Then call both the recursive and the iterative functions. Compare how long it takes to find the desired Fibonacci number. Note how much faster the iterative version becomes (compared to the recursive version) as you ask for higher numbers.

```
(* Iterative Fibonacci function. Function works by
   storing values of the two most recent Fibonacci numbers,
   since these are used to compute the next one.
*)
FUNCTION IterFib ( Val : INTEGER) : LONGINT;
VAR Smaller, Larger, RunningFib : LONGINT;
    Count : INTEGER;
BEGIN
  (* Fib ( 1) and Fib ( 2) = 1 *)
  IF Val < 3 THEN
    IterFib := 1
  (* Fib ( 3) = 2 *)
  ELSE IF Val = 3 THEN
    IterFib := 2
  ELSE   (* If Val > 3 *)
  BEGIN
    (* Smaller contains Fib ( n - 2);
       Larger contains Fib ( n - 1);
    *)
    Smaller := 1;
    Larger := 2;
    Count := 4;
```

```
      REPEAT
        (* RunningFib is the current Fibonacci number *)
        RunningFib := Smaller + Larger;
        (* After computing a new number, the values of
           Smaller and Larger change.
        *)
        Smaller := Larger;
        Larger := RunningFib;
        INC ( Count);
      UNTIL Count > Val;
      IterFib := RunningFib;
    END;   (* IF Val > 3 *)
  END;  (* FN IterFib *)
```

This function works by storing the two most recent Fibonacci numbers as intermediate values, and using these to compute each successive Fibonacci number. As each new number is computed, that number becomes **Fib(N-1)** for the next value to be computed.

The listing itself is much longer than the recursive solution. However, it is also much faster. Whereas the recursive version takes almost two million calls to compute **Fib(30)**, the iterative version manages it in a few dozen additions.

Despite its inefficiencies, recursion is a very powerful technique. You'll use it in later chapters, to build a data structure called a list.

Summary

This chapter covered some miscellaneous information about procedures and functions. Some of this information (such as the scope and visibility rules) is also useful for understanding other aspects of the language (such as the use of variable names in different parts of the program.)

The chapter also discussed recursion, a programming technique in which a routine calls itself to accomplish part of its task.

In the next chapter, you'll learn about CodeWarrior Pascal's more advanced data types.

7

CodeWarrior Pascal's Structured Data Types

The data types you've learned about so far have been used to store various types of information. However, each variable could store exactly one item of information. (The one exception to this restriction is the **STRING** type, which you can view as a single string or as the individual characters that make up the string.)

In this chapter, you'll learn about some other data types, which are used for storing multiple items of information in the same variable. These data types — **ARRAY**, **RECORD**, and **SET** — are known as *structured*, or *aggregate*, types, since variables or values of these types are actually built up of components.

For example, an array consists of one or more components (cells), each of which contains the same type of information. A record also consists of one or more components (fields), each of which may contain the same or a different type of information. Finally, a set is an unordered collection of values (elements), all of the same type. You must specify the details of these components when you define your types.

You can use these structured types as building blocks for creating data types that exactly suit the needs of your problem. For example,

- If you need to store the grade point averages of an entire class, you can store them in an array of **REAL**s.
- If you need to store a profile about a student (for example, name, grade point average, major, etc.), you can store this information in a record.
- If you need to store profiles for an entire class, you can store them in an array, each of whose cells is a record. (Each cell in such an array *is* of the same type. The type just happens to be a complex one.)
- If you need to keep track of the courses a student has taken, you can use a set to represent this course list.

- If you need to associate this information with a student's profile, you can create a record that has both a profile and a course set as components.
- If you need to associate student profiles and course lists for an entire class, you can create an array whose cells consist of profile-course list records.
- If you need to build a shelf to hold your course notes and handouts, you need to use an erector set or go to a home supply store to get the needed materials. Pascal is a powerful programming language, but even it has its limitations.

This chapter also includes a brief introduction to the **OBJECT** data type used in object-oriented programming. You'll learn more about this powerful programming technique in Chapter 11.

Definitions and Declarations

When you declare a variable as being of a simple type, the compiler knows exactly what you mean. For each simple type, the compiler knows the operations allowed and the storage to be allocated. Things are a bit different when you're working with structured types, however.

Structured types are really building blocks. What you get are several types of blocks, each of which can be of just about any type.

- homogeneous blocks (arrays)
- heterogeneous blocks (records), and
- blocks whose contents can increase or decrease, as needed (sets)
- record-like blocks that can inherit components from other blocks (objects)

Thus, when you create an aggregate type, you can vary many things in your description. For example, you can specify the number of elements the new type can hold as well as the type of elements to be stored in the cells.

Because these blocks are really generic components, you need to provide a description of a particular structured type before you can declare variables of that type. Syntactically, you provide such descriptions as **TYPE** definitions in the declaration section of the program or routine.

You'll see specific examples in the next few sections.

TIP: You need to describe a new structured data type before you can declare variables of that type. You provide such a description when you define the data type in a program or routine.

The ARRAY Type

Suppose you want to test CodeWarrior Pascal's random number generator, to see whether all digit pairs are equally likely to end the integers generated. One way to do this would be to count the number of times each pair of digits (that is, 00 through 99) ends random integers.

To represent the 100 frequencies that result, you could declare 100 different variables, for example, **F0**, F1, **F2**, and so on, through **F99** — a tedious chore, to say the least.

After you had declared these variables, you would need to do just as much work to modify the frequencies after each integer is generated. For example, you would need a **CASE** statement with 100 possible values to make the required changes. The following listing shows what the shell of such a statement would look like.

```
CASE Remainder OF
  0 : INC ( F0);
  1 : INC ( F1);
  2 : INC ( F2);
  (*
     and so on, to
  *)
 98 : INC ( F98);
 99 : INC ( F99);
END; (* CASE of Remainder *)
```

This looks like a lot of work, and it is! Fortunately, Pascal provides the **ARRAY** type to handle such situations. An *array* is an aggregate variable whose components are multiple values, all of the same type. These individual components are known as *cells* of the array. Each cell has an *index* associated with it — to identify the cell.

In Pascal, an array must have a fixed number of elements. The cell indexes for these elements must be between explicitly specified starting and ending indexes (bounds).

Defining and Declaring an Array

For example, the data for the random digits problem can be represented in a 100-element array, each of whose cells will contain an integer. The following program shows how to define, declare, and use such an array. Run the program to see a sample outcome.

Listing 7.1

```
(* Program to illustrate use of ARRAYs. Program generates random
   integers, and counts the frequency with which each pair
   of digits terminates these numbers.
*)

PROGRAM TestDigits;
USES Windows;

CONST MaxTrials = 5000;
(* Describe the data type to be used *)
TYPE Remainders = ARRAY [ 0 .. 99] OF INTEGER;
VAR Freqs : Remainders;
    Index, Result, RandSeed : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Initialize the cells of Vals.
     Note: this is a call by reference.
  *)
  PROCEDURE InitRemainders ( VAR Vals : Remainders;
                             InitValue : INTEGER);
  VAR Index : INTEGER;
  BEGIN
    FOR Index := 0 TO 99 DO
      (* Initialize the IndexTH cell *)
      Vals [ Index] := InitValue;
  END; (* InitRemainders *)

  (* Display Remainders array — 4 cells per line.
     Note: this is a call by value.
  *)
  PROCEDURE DispRemainders ( Vals : Remainders);
  VAR Index : INTEGER;
  BEGIN
    (* all values are written, even though loop counter only
       goes to 24 — because four values per line are written.
    *)
    FOR Index := 0 TO 24 DO
    BEGIN
      WRITE ( '[':5, Index:3, ']: ', Freqs [Index]:3);
      WRITE ( '[':5, Index + 25:3, ']: ', Freqs [Index + 25]:3);
      WRITE ( '[':5, Index + 50:3, ']: ', Freqs [Index + 50]:3);
      WRITE ( '[':5, Index + 75:3, ']: ', Freqs [Index + 75]:3);
      WRITELN;
    END; (* FOR Index := 0 TO 24 *)
  END; (* DispRemainders *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN (* Main *)
  GetInt ( 'Integer for random number generator', RandSeed);
  SRAND ( RandSeed);
  InitRemainders ( Freqs, 0);

  (* generate the values, and categorize the results *)
  FOR Index := 1 TO MaxTrials DO
  BEGIN
```

```
    Result := RAND MOD 100;
    (* E.g., if Result = 39, then increment Freqs [ 39] by 1 *)
    Freqs [ Result] := Freqs [ Result] + 1;
  END;

  DispRemainders ( Freqs);
  AwaitUser;
END.
```

Recall that you first need to define the array type you intend to use. This is done in the **TYPE** definition statement:

```
TYPE Remainders = ARRAY [ 0 .. 99] OF INTEGER;
```

This statement describes a new type of variable, which will be named **Remainders**. The type is built out of familiar components (integers). In particular, **Remainders** is defined as an **ARRAY**.

Recall that an array's components are all of the same type. As part of the definition, you must specify this *base type*. For **Remainders**, the base type is **INTEGER**. This means that each cell of the array will be allocated two bytes, and will contain an **INTEGER** value. Essentially, each cell is an **INTEGER** variable, except that it has been subsumed in a larger structure. As a variable, a cell can be used wherever a variable of that type is allowed.

You also need to specify a fixed size for the array. The information within the square brackets (**[0 .. 99]**) represents the range of the indexes that should be associated with the array's cells. In this case, the definition says that the cells are to be “named" **0** through **99**, and indirectly tells the compiler that the array has 100 cells.

Once you've defined the new type you can declare variables of that type. The first **VAR** declaration line shows how to do this.

The first declaration will have the reserved word **VAR** before it.

Accessing Individual Array Cells

In the **TestDigits** program, the value of one particular cell of the array is increased each time through the main program's **FOR** loop. The assignment statement that accomplishes this shows how to refer to specific cells.

To access the appropriate cell, specify the variable name (**Freqs**), followed by the cell's index within square brackets. In the program, the remainder from the **MOD** operation represents the index of the cell whose frequency count needs to be increased. Thus,

```
Freqs [ Result]
```

specifies the cell whose index corresponds to the current value of **Result**.

The right hand side of the assignment statement increases the value of this cell by 1. Note that the 1 is outside the square brackets. The following statement accomplishes the same increment:

```
INC ( Freqs [ Result]);
```

Freqs[Result] is a memory location that contains an **INTEGER** value. As such, it is a valid argument for the **INC** procedure.

Cell Index versus Cell Value

Don't confuse a cell's index with the value stored in the cell. The index is part of the cell's name. Thus, **Freqs[5]** represents a specific memory location (which contains an **INTEGER**).

Cell **Freqs[5]** has a value stored in it, which represents the frequency with which the random numbers ended in 05 (such as 3905, 105). Conversely, this value is stored at the location that corresponds to **Freqs[5]**.

To see more directly the difference between cell index and cell value, compare the two assignment statements in the following listing. Assume **Freqs[5]** has the value 47, and **Freqs[6]** has the value 62.

```
(* Assume Freqs [ 5] = 47; Freqs [ 6] = 62 *)

Freqs [ 5] := Freqs [ 5] + 1;
Freqs [ 5] := Freqs [ 5 + 1];
```

After the first assignment, the value of **Freqs[5]** will be 48 — that is, 1 more than before the assignment. After the second assignment, **Freqs[5]** will have the value 62 — that is the same value as **Freqs[6]**.

WARNING! Don't confuse a cell's index and its value. The index serves to identify the cell by its position in the array. The cell's value represents the contents of the memory location allocated for the cell.

Representing an Array

When you declare an array variable, the compiler allocates enough memory to store as many values of the base type as there are cells in the array. Thus, **Freqs** is allocated 200 bytes of storage, as you'll see if you add the following statement to the main program of **TestDigits**:

```
WRITELN ( 'Freqs takes ', SIZEOF ( Freqs), ' bytes of stor-
age');
```

In memory, the cells of an array are stored in consecutive storage areas. Each cell is allocated enough space to store a value of the array's base type.

The index values are not stored in memory. Rather, the compiler just uses them when computing the memory location for the cell. Only the frequency values (48, 44, 50, etc.) are stored in memory.

Rules for Arrays

An array consists of cells. The cells can contain just about any type of information, which is known as the array's base type. However, each cell in the array must have the same base type. An array's size will be determined by the base type and the number of elements. This size cannot be larger than 64K bytes. An array also cannot contain an array identical to itself as a component.

You can use any ordinal type (except **LONGINT** or a subrange of this) to specify cell indexes. To indicate both the number of cells and the indexes to be associated with those cells, specify a starting index value and an ending index value, separated by double dots (..).

The ending index value must be greater than the starting value. The compiler assumes the array contains as many cells as there are values between the starting and ending values (including the endpoints). For example, in the **TestDigits** program, there are 100 values between 0 and 99.

The following are all valid array definitions, using different types to specify index values:

```
TYPE MyOrdinalType = ( one, two, three, four, five, six,
                       seven, eight, nine, ten, eleven,
                       twelve, thirteen, fourteen);

      (* INTEGER indexes: array has 14 cells *)
      WordArray = ARRAY [ -5 .. 8 ] OF WORD;

      (* CHAR indexes: array has 20 cells *)
      CharIndexArray = ARRAY [ 'a' .. 't'] OF REAL;

      (* MyOrdinalType indexes: array has 14 cells *)
      MyTypeIndexArray = ARRAY [ one .. fourteen] OF CHAR;

      (* BOOLEAN indexes: array has 2 cells *)
     BoolIndexArray = ARRAY [ FALSE .. TRUE] OF MyOrdinalType;

      (* each "cell" is a string; array has 11 cells *)
      StringArray = ARRAY [ -5 .. 5] OF STRING;
```

Arrays as Parameters

The **TestDigits** program shows that you can pass array variables as arguments to procedures and functions. These arguments can be passed by value (for example, in **DispRemainders**) or by reference (for example, in **InitRemainders**).

Inside the called routine, the cells of the array are accessed using the same index values as in the calling routine, but using the local name for the array (**Vals** in both **InitRemainders** and **DispRemainders**).

Procedure **DispRemainders** displays the contents of the array, four cells per line. Note the indexing being done in **DispRemainders**. The values written on a particular line are those that belong to cells with indexes *X, X + 25, X + 50,* and *X + 75*.

The following listing shows sample results from the program:

```
[  0]:  48    [ 25]:  61    [ 50]:  33    [ 75]:  54
[  1]:  44    [ 26]:  46    [ 51]:  39    [ 76]:  48
[  2]:  50    [ 27]:  56    [ 52]:  56    [ 77]:  36
[  3]:  52    [ 28]:  36    [ 53]:  61    [ 78]:  48
[  4]:  49    [ 29]:  39    [ 54]:  43    [ 79]:  49
[  5]:  46    [ 30]:  56    [ 55]:  56    [ 80]:  61
[  6]:  33    [ 31]:  53    [ 56]:  46    [ 81]:  53
[  7]:  36    [ 32]:  40    [ 57]:  56    [ 82]:  35
[  8]:  51    [ 33]:  66    [ 58]:  57    [ 83]:  44
[  9]:  45    [ 34]:  48    [ 59]:  56    [ 84]:  49
[ 10]:  34    [ 35]:  60    [ 60]:  59    [ 85]:  55
[ 11]:  49    [ 36]:  57    [ 61]:  42    [ 86]:  55
[ 12]:  61    [ 37]:  47    [ 62]:  38    [ 87]:  50
[ 13]:  45    [ 38]:  60    [ 63]:  51    [ 88]:  42
[ 14]:  53    [ 39]:  53    [ 64]:  49    [ 89]:  50
[ 15]:  55    [ 40]:  46    [ 65]:  47    [ 90]:  57
[ 16]:  41    [ 41]:  58    [ 66]:  57    [ 91]:  45
[ 17]:  40    [ 42]:  60    [ 67]:  51    [ 92]:  54
[ 18]:  53    [ 43]:  49    [ 68]:  55    [ 93]:  40
[ 19]:  42    [ 44]:  46    [ 69]:  56    [ 94]:  56
[ 20]:  57    [ 45]:  49    [ 70]:  43    [ 95]:  46
[ 21]:  55    [ 46]:  56    [ 71]:  52    [ 96]:  43
[ 22]:  58    [ 47]:  54    [ 72]:  61    [ 97]:  62
[ 23]:  49    [ 48]:  42    [ 73]:  51    [ 98]:  57
[ 24]:  48    [ 49]:  60    [ 74]:  52    [ 99]:  52
```

This output indicates, for example, that the value of cell **Freqs[74]** was 52 after the program finished executing. This means that the digit pair 74 ended 52 of the 5000 (**MaxTrials**) random values generated. Similarly, the value of **Freqs[93]** was 40. Notice that **Freqs[74]** is actually the 75th cell, since the first cell has index 0.

Operations on Arrays

If you have two array variables of exactly the same type, you can make the two arrays identical with a single assignment statement. For example, suppose **Freqs1** and **Freqs2** are each variables of type **Remainders** (as in the **TestDigits** program). The following assignment is syntactically valid:

```
Freqs2 := Freqs1;
```

The effect of this assignment is equivalent to the following loop

```
FOR Index := 0 TO 99 DO
  Freqs2 [ Index] := Freqs1 [ Index];
```

Beyond this total assignment operation, arrays are accessed and manipulated cell by cell. The operations that can be applied to individual cells depend on the base type of the cells.

One More Example

Let's look at one more example, which illustrates some similarities between arrays and the mathematical concept of a vector. A *vector* is an ordered collection of values. Generally these values represent information that is related.

For example, you would use a vector to represent a point in space. For each point you would have a value representing the point's position in each of the three spatial dimensions (x, y, and z).

To represent this information in an array, you might use a definition such as the one shown in the following program:

Listing 7.2

```
(* Program illustrating use of arrays as vectors *)

PROGRAM DistanceDemo;

     (* Define a type to represent the three dimensions.
        NOTE: the values are NOT equivalent to `x,' `y,' and `z'
     *)
TYPE Axes = ( x, y, z);
     (* A vector contains three values —
        corresponding to the three dimensions.
     *)
     Vector = ARRAY [ x .. z] OF REAL;
VAR PointA, PointB : Vector;
    Dist : REAL;

   (* Wait for user to press ENTER *)
```

```
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

  (* Get three coordinate values from the user *)
  PROCEDURE GetPoint ( VAR Pt : Vector);
  BEGIN
    GetReal ( 'x-value?', Pt [ x]);
    GetReal ( 'y-value?', Pt [ y]);
    GetReal ( 'z-value?', Pt [ z]);
  END; (* GetPoint *)

  (* Compute the distance between two points in space.
     Function uses the following formula:
     Dist := SQRT ( XDiff^2 + YDiff^2 + ZDiff^2),
     where XDiff := PtA[ x] - PtB [ x], etc.
  *)
  FUNCTION Distance3D ( PtA, PtB : Vector) : REAL;
  VAR Sum : REAL;
      Count : Axes;
  BEGIN
    Sum := 0.0;
    FOR Count := x TO z DO
      Sum := Sum + SQR ( PtA [ Count] - PtB [ Count]);
    Distance3D := SQRT ( Sum);
  END; (* FN Distance3D *)

BEGIN
  WRITELN ( 'Enter point information.');
  WRITELN ( ' To STOP, make both points equal.');
  WRITELN;
  REPEAT
    WRITELN ( 'Point A:');
    GetPoint  ( PointA);
    WRITELN ( 'Point B:');
    GetPoint  ( PointB);
    Dist := Distance3D ( PointA, PointB);
    WRITELN ( 'Distance between points is ', Dist:10:3);
   (* termination condition doesn't test for strict
      equality to avoid differences due to rounding errors.
  *)
  UNTIL Dist <= 0.0;
  AwaitUser;
END.
```

The program computes the distance between any two points that you specify. To specify a point, you need to provide, x, y, and z-coordinates.

Strings

In Chapter 3, you learned that a **STRING** is not a simple type, although you can often treat it as one. Also, in an earlier listing, you learned that an expression such as

```
MyString [ 6]
```

represents the sixth character in the string **MyString**.

It may be no surprise to learn that CodeWarrior Pascal's **STRING** type is represented as an array of **CHAR**. By default, this is an array of 255 elements. You can, however, specify a maximum length for a string. For example,

```
STRING [ 45]
```

specifies a string with at most 45 characters.

Traditionally, CodeWarrior Pascal has represented strings as an array of characters. This array uses index values 0 .. 255 by default. The contents of cells 1 through 255 are the characters (if any) in the string. The value stored in cell 0 represents the length of the string — that is, the number of characters actually stored in a particular string. Because of the way string length is handled, this representation is limited to at most 255 characters (plus a 256th for length information).

The following program shows how to use **STRING** variables and how to access and work with individual characters in the string. The program reads a string entered by the user, and then sorts the characters in the string.

Listing 7.3

```
(* Program to illustrate use of strings.
   Program sorts strings entered by the user.
*)

PROGRAM StringBubble;

CONST EmptyString = '';
VAR   CurrStr : STRING;
      Count : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Info : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Info);
  END; (* GetString *)

  (* Sort characters in a string by letting the "largest"
     remaining character work its way to the top of the
     array on each pass through the array.
  *)
  PROCEDURE StrBubbleSort ( VAR Info : String);
  VAR Temp : CHAR;
      Low, High, Top, Size : INTEGER;
  BEGIN
    Size := LENGTH ( Info);
    Top := Size + 1;
    WHILE ( Top > 1) DO
    BEGIN
      Low := 1;
      High := 2;
      WHILE High < Top DO
      BEGIN
        (* if lower string needs to move upward in the array *)
        IF Info [ Low] > Info [ High] THEN
        BEGIN
          Temp := Info [ Low];
          Info [ Low] := Info [ High];
          Info [ High] := Temp;
        END; (* If lower string needs to move upward *)
        Low := Low + 1;
        High := High + 1;
      END; (* WHILE High < Top *)
      Top := Top - 1;
    END; (* WHILE Top > 1 *)
  END;  (* StrBubbleSort *)

BEGIN
  (* Get the lines to be sorted *)
  Count := 0;
  WRITELN ( 'Enter your strings.');
  WRITELN ( ' To STOP, enter an empty string.');
  WRITELN;
  REPEAT
```

```
    GetString ( ':', CurrStr);
    Count := Count + 1;
    WRITELN ( CurrStr);
    StrBubbleSort ( CurrStr);
    WRITELN ( CurrStr);
  UNTIL CurrStr = EmptyString;
  WRITELN ( Count:3, ' lines read');
  AwaitUser;
END.
```

The program uses a bubblesort algorithm to sort the letters in the strings entered by the user. Procedure **StrBubbleSort** is quite similar to **BubbleSort** in Chapter 2. The major difference is that **StrBubbleSort** works with characters in individual strings.

The following listing shows some sample output from the program. Notice that blanks are sorted along with the alphabetic characters.

```
: aardvark
aardvark
aaadkrrv

: ant
ant
ant

: banana slug
banana slug
 aaabglnnsu

: wapiti
wapiti
aiiptw

: wolf
wolf
flow

: once upon a time, there was a very long sentence
once upon a time, there was a very long sentence
         ,aaacceeeeeeeeghilmnnnnnoooprrsstttuvwy
```

```
7 lines read
```

Notice that you can declare **STRING** variables without first doing a **TYPE** definition. This is because the **STRING** type is predefined for you in CodeWarrior Pascal.

Multidimensional Arrays

An array's base type can be just about any kind of information — including another array. For example, suppose you want to record the highest temperature during the day and the highest at night. Suppose also that you want to record this information for an entire month.

If you were recording this information by hand, you might use a two column table, with one column for
Day and the other for Night. The rows in this table would correspond to the individual days. This table of information has 62 different cells, grouped into 31 pairs. Each row of the table represents a pair of values. (You would probably use a third, Date, column to represent the indexes. This column is not part of the array.)

You can think of this data structure as 62 individual cells, or as 31 rows of 2 cells each. In the latter case, you can consider each row a 2-element array.

There are actually several ways you can define such an array in Pascal, depending on whether you also want the component (2-element) array as a separate type. The following listing shows three different ways of defining the same kind of array:

```
(* make the 2-element array a separate, named data type *)
TYPE DailyTemps = ARRAY [ 1 .. 2] OF REAL;
     TempData1 = ARRAY [ 1 .. 31] OF DailyTemps;

     (* describe an array built of an anonymous array type *)
     TempData2 = ARRAY [ 1 .. 31] OF ARRAY [ 1 .. 2] OF REAL;

     TempData3 = ARRAY [ 1 .. 31, 1 .. 2] OF REAL;
```

A **TempData1** array consists of 31 cells of type **DailyTemps**. The preceding **TYPE** definition tells you that this **DailyTemps** is a two-cell array of **REAL**.

By providing a name for this component type, you've made it possible to declare variables of that type. For example, the following declaration would be valid, given the preceding definitions:

```
VAR TodaysTemps : DailyTemps;
```

The definition for **TempData2** also specifies that it is a 31-element array whose cells are two-element arrays. In contrast to **TempData1**, however, this component type is anonymous. No name is provided for the type; rather, the type is simply described "on the fly." As a consequence, you can't declare any variables of this type.

Finally, the definition for **TempData3** simply specifies that it's a two-dimensional array, whose first dimension has 31 elements and whose second has 2 elements.

Accessing Elements in Multidimensional Arrays

The table discussed in the previous section resembles a *matrix* in mathematics. This is a structure consisting of rows of values. Each row is a vector with 1 or more values. Every row in a matrix has the same number of elements. The rows are arranged vertically, so that the first element of every vector is in the same (first) column of the matrix, each vector's second element is in the second column, and so forth.

If you think of your table as a matrix, it will have 31 rows, each with two columns. This idea will help you to see how to access individual elements in such an array.

Run the following program to see how such cells are accessed.

Listing 7.4

```
(* Program to illustrate use of two-dimensional arrays *)

PROGRAM MatrixDemo;

TYPE Matrix = ARRAY [ 1 .. 5, 1 .. 3] OF INTEGER;
VAR SampleMatrix : Matrix;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Put a value in each cell of Vals *)
  PROCEDURE FillMatrix ( VAR Vals : Matrix);
  VAR Row, Column : INTEGER;
  BEGIN
    FOR Row := 1 TO 5 DO
      FOR Column := 1 TO 3 DO
        (* To specify a single cell in a matrix, specify
           the row and column within the square brackets.
        *)
        Vals [ Row, Column] := 10 * Row + Column;
  END; (* FillMatrix *)

  (* Display the contents of Vals *)
  PROCEDURE DispMatrix ( Vals : Matrix);
  VAR Row, Column : INTEGER;
  BEGIN
    WRITELN ( 'Col 1':13, 'Col 2':8, 'Col 3':8);
    FOR Row := 1 TO 5 DO
    BEGIN
      WRITE ( 'Row ', Row);
      FOR Column := 1 TO 3 DO
        WRITE ( Vals [ Row][ Column]:8);
      WRITELN;
    END;
  END; (* DispMatrix *)

BEGIN  (* Main program *)
  FillMatrix ( SampleMatrix);
  DispMatrix ( SampleMatrix);
  AwaitUser;
END.
```

The program defines a type named **Matrix**, which describes a two dimensional array with 5 rows and 3 columns. Such an array has 15 (5 * 3) elements, each of which is an **INTE-**

GER.

The program uses two procedures — **FillMatrix** and **DispMatrix** — to do its work. Note that, like a one-dimensional array or a simple type, a two-dimensional array can be passed as a parameter.

The array is passed by reference to **FillMatrix**, since this procedure is supposed to put values in the cells. **DispMatrix**, on the other hand, does not change anything in the array, so a copy will do just as well. The parameter is passed by value in that case.

The program produces the following output:

```
          Col 1    Col 2    Col 3
Row 1       11       12       13
Row 2       21       22       23
Row 3       31       32       33
Row 4       41       42       43
Row 5       51       52       53
```

Procedure **FillMatrix** accesses the individual cells by using one **FOR** loop nested inside another. The outer loop, whose values change more slowly, represents the first (row) dimension; the inner loop represents the second (column) dimension.

The higher dimensions change more quickly. Thus, to visit each cell of a **Matrix**, you would first visit each cell in row 1. That is, you would visit columns 1 through 3 in row 1 (cells [1,1], [1,2], and [1,3]). Then you would visit each successive row until you had visited all three columns in row 5.

The actual work in **FillMatrix** is done by the single statement in the inner **FOR** loop. This statement assigns a value to a particular cell. The rest of the loops' activity is administrative.

Notice how the cell is specified. To select a cell, indicate the row and then the column within square brackets. The two indexes are separated by commas. Thus, the statement

```
Vals [ 4, 2] := 10 * 4 + 2;
```

would assign the value 42 to the second cell in the fourth row.

Procedure **DispMatrix** shows a slightly different syntax for specifying a cell. Instead of putting both indexes within one set of brackets, you can use separate brackets for each index. Thus, the statement

```
Vals [ 4][ 2] := 10 * 4 + 2;
```

also assigns 42 to the second element in the fourth row. The comma syntax is easier and more common.

Accessing Component Arrays

In the **MatrixDemo** program, you accessed individual (**INTEGER**) cells of the array. However, remember that you can think of a two-dimensional array as one with arrays as elements.

Is there any way to access an entire row in the matrix? The following procedure shows how to assign values to a **Matrix** variable, an entire row at a time. The listing is not a stand-alone program, so don't try to compile it as such.

```
 (* Set entire rows of the matrix to new values.
    Row1 := Row5;  Row2 := Row4;  Row3 := Row3;
    Row4 := Row2;  Row5 := Row1;
 *)
 PROCEDURE ShuffleMatrix ( VAR Vals : Matrix);
 VAR Row : INTEGER;
 BEGIN
   FOR Row := 1 TO 5 DO
     (* To access an entire row, omit the column index *)
     Vals [ Row] := Vals [ 5 - Row + 1];
 END; (* ShuffleMatrix *)
```

Procedure **ShuffleMatrix** assigns the array stored as row 5 to the array stored as row 1. Row 2 is assigned the contents of Row 4, and so on. In each of these assignments, the three cells in a row are assigned values simultaneously.

To indicate how the assignment should work, only the row dimension is included when specifying the variable. This tells the program to work with the base type for a row, which will be an array.

If you add the **ShuffleMatrix** procedure to the **MatrixDemo** program, and then add calls to **ShuffleMatrix** and **DispMatrix** at the end of the main program, you get the following output.

```
        Col 1   Col 2   Col 3
Row 1      11      12      13
Row 2      21      22      23
Row 3      31      32      33
Row 4      41      42      43
Row 5      51      52      53
        Col 1   Col 2   Col 3
Row 1      51      52      53
Row 2      41      42      43
Row 3      31      32      33
Row 4      41      42      43
Row 5      51      52      53
```

If you're not sure why the array's rows haven't simply been reversed, think about the contents of **Vals[2]** by the time this array is assigned to **Vals[4]**.

String Arrays

An array of type **STRING** actually is another example of a two-dimensional array. The following program shows such an array:

Listing 7.5

```
(* Program to illustrate use of string arrays *)

PROGRAM MenuDemo;

CONST MaxMenu = 20;
TYPE StrArray = ARRAY [ 1 .. MaxMenu] OF STRING [ 80];
VAR Menu : StrArray;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Initialize the menu to the available choices.
      Cells for which no choices were provided are
      initialized as empty strings.
  *)
  PROCEDURE InitMenu ( VAR Menu : StrArray);
  VAR Index : INTEGER;
  BEGIN
    (* the first three cells get actual values in the example *)
    Menu [ 1] := 'Choice1';
    Menu [ 2] := 'Choice2';
    Menu [ 3] := 'Choice3';
    (* the remaining cells get null strings *)
    FOR Index := 4 TO MaxMenu DO
      Menu [ Index] := '';
  END;  (* InitMenu *)

 (* Display the contents of the menu array.
    NrChoices represents the number of Menu's cells
    that have choices (as opposed to empty strings)
  *)
  PROCEDURE ShowMenu ( VAR Menu : StrArray;
                       NrChoices : INTEGER);
  VAR Index : INTEGER;
  BEGIN
    (* To prevent procedure from reading past end of array *)
    IF NrChoices > MaxMenu THEN
      NrChoices := MaxMenu;
    FOR Index := 1 TO NrChoices DO
      WRITELN ( Menu [ Index]);
    WRITELN;
    WRITE ( 'Your Choice? (1 TO ', NrChoices, ') ');
  END; (* ShowMenu *)

BEGIN
  InitMenu ( Menu);
  ShowMenu ( Menu, 3);
```

```
   AwaitUser;
END.
```

This program defines two procedures that will be useful later in the book. The procedures are used to initialize and display a menu of choices for the user. In later programs, you'll use such menus to interact with the user.

Notice that only the row dimension is used when manipulating elements of the **Menu** array. This is because strings are generally treated as if they were a simple variable — that is, as a unit, rather than as a sequence of individual characters.

You can still access individual character cells. To do this, simply specify the second index as well as the row. For example, you would refer to the sixth character in the third string ('e') as:

```
Menu [ 3, 6]
```

Notice that **ShowMenu**'s first parameter is declared as a **VAR** parameter, even though the array's contents are not changed. This is common when working with arrays. If you pass an array by value, a copy of the entire array must be made for the called routine's local environment. For an array such as **Menu**, that would be over 5000 bytes. That can be a significant amount of memory to lose — even temporarily -- if you've got a large program that uses lots of memory for other things.

When you pass **Menu** by reference, you make it unnecessary to copy the array to the routine's local environment. Rather, the routine will assume that the array is at the location of the argument specified in the call. This saves time and storage, both of which can make a difference in a large program. Of course, you must be careful not to change values inadvertently.

Note that **ShowMenu** first checks that **NrChoices** does not exceed the **StrArray**'s bounds. If you try to read past the end of a string (or any array), the program terminates with a run-time error.

WARNING! Be careful not to access array elements beyond the bounds of an array. If your program does so, it will end with a run-time error.

Three-Dimensional Arrays

Earlier, you learned how to define a matrix, an array of arrays. You can also have an array of matrices. That is, you can have arrays with three or more dimensions. For example, the following program shows how to define, declare and use a three-dimensional array:

Listing 7.6

```
(* Program to illustrate use of three-dimensional arrays *)

PROGRAM HyperMatrixDemo;

TYPE HyperMatrix = ARRAY [ 1 .. 3, 1 .. 4, 1 .. 5] OF INTEGER;
VAR Sample : HyperMatrix;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Put a value in each cell of Vals *)
  PROCEDURE FillHyperMatrix ( VAR Vals : HyperMatrix);
  VAR Plane, Row, Column : INTEGER;
  BEGIN
    FOR Plane := 1 TO 3 DO
      FOR Row := 1 TO 4 DO
        FOR Column := 1 TO 5 DO
          (* To specify a single cell in a HyperMatrix, specify
             the row and column within the square brackets.
          *)
          Vals [ Plane, Row, Column] := 100 * Plane +
                                        10 * Row + Column;
  END; (* FillHyperMatrix *)

  (* Display the contents of Vals *)
  PROCEDURE DispHyperMatrix ( Vals : HyperMatrix);
  VAR Plane, Row, Column : INTEGER;
  BEGIN
    FOR Plane := 1 TO 3 DO
    BEGIN
      WRITELN ( 'Plane ':10, Plane);
      WRITELN ( 'Col 1':13, 'Col 2':8, 'Col 3':8,
                'Col 4':8, 'Col 5':8);
      FOR Row := 1 TO 4 DO
      BEGIN
        WRITE ( 'Row ', Row);
        FOR Column := 1 TO 5 DO
          WRITE ( Vals [ Plane][ Row][ Column]:8);
        WRITELN;
      END;  (* FOR Row *)
    END; (* FOR Plane *)
  END; (* DispHyperMatrix *)

  (* Set entire matrices of the hypermatrix to new values.
     Plane1 := Plane3;  Plane2 := Plane2;  Plane3 := Plane1;
  *)
  PROCEDURE ShuffleHyperMatrix ( VAR Vals : HyperMatrix);
  VAR Plane : INTEGER;
  BEGIN
    FOR Plane := 1 TO 3 DO
      (* To access an entire plane row,
         omit the row and column indexes
      *)
      Vals [ Plane] := Vals [ 3 - Plane + 1];
  END; (* ShuffleHyperMatrix *)

BEGIN
  FillHyperMatrix ( Sample);
  DispHyperMatrix ( Sample);
  ShuffleHyperMatrix ( Sample);
  DispHyperMatrix ( Sample);
  AwaitUser;
END.
```

This program does the same thing as **MatrixDemo**, except that this version works with arrays of matrices. **FillHyperMatrix** shows one way of specifying individual (**INTEGER**) cells in the **HyperMatrix**; DispHyperMatrix shows a second way.

Procedure **ShuffleMatrix** shows how to assign an entire matrix of values at once — by leaving off the indexes for both rows and columns. To access an entire row of a matrix at once, omit only the column index.

Arrays: A Final Point ... and Another Data Type

Before moving on from arrays, we need to look at one more feature of many array definitions. To do that, however, we need to discuss a Pascal data type that has been mentioned only in passing so far.

The Subrange Type

Pascal has a simple type that was not discussed in Chapter 3. This type is simple in that each variable contains only a single value. However, the *subrange* type is actually derived from another simple type, and represents a limited range of values of the source type.

Because it is a derived type, a subrange type must be defined before you can declare a variable of that type. This is because the compiler needs to know the range of values included in the subrange type.

For example, suppose you wanted to use only single-digit integers — that is, values from 0 through 9. You could accomplish this as follows:

```
TYPE SingleDigit = 0 .. 9;
VAR DigitVal : SingleDigit;
```

The first statement specifies that variables of type **SingleDigit** can take on values from 0 through 9. Notice that there are no parentheses around the subrange specifiers. Parentheses are needed only when you are telling the compiler about *new* values. The compiler already knows about whole numbers, so it can interpret the **0 .. 9** properly.

Subrange types are most commonly used to specify array indexes, as in the following listing:

```
TYPE SingleDigit = 0 .. 9;
     FaceValue = 2 .. 14;
     (* first define an enumerated type *)
     CardSuit = ( club, diamond, heart, spade);
     (* specify a subrange of that type *)
     RedSuit = diamond .. heart;

     SmallVal = ARRAY [ SingleDigit] OF REAL;
     (* arrays whose cells are TRUE if the card
        is in the hand, FALSE otherwise.
     *)
     RedCards = ARRAY [ FaceValue, RedSuit] OF BOOLEAN;
     AnyCards = ARRAY [ FaceValue, CardSuit] OF BOOLEAN;

VAR DigitData :  SmallVal;
    RedHand : RedCards;
    Hand : AnyCards;
```

First, look at the definitions for **SingleDigit** and **SmallVal**. The subrange type is defined, as in the earlier listing. Once this is done, the compiler knows that **SingleDigit** is another way of saying **0 .. 9**. Thus, when it sees this identifier as the range specification between the square brackets, the compiler simply substitutes the **0 .. 9**. The result is that you've defined a 10-element array, with indexes ranging from 0 through 9.

The other array definitions are a bit more involved. Notice the enumerated type definition, for **CardSuit**. This definition specifies that the type **CardSuit** can take on the values **club**, diamond, **heart**, and **spade**.

Once you've defined this type, you can specify subranges based on the **CardSuit**. For example, **RedSuit** specifies a subrange of **CardSuit** values — namely, the two middle values. Notice that the two dots (..) are still used between the starting and ending values, even if there is no intermediate value.

Notice also that there are parentheses around the original list of values when defining **CardSuit**. There are no parentheses around the subrange specification, since the compiler already knows what to do with the values.

After these definitions, you'll find two definitions for two-dimensional arrays. The first dimension in each case is the **FaceValue** subrange type. This type takes on values between 2 and 14, inclusive.

The second dimension for the **RedCards** array is of type **RedSuit**. This means that the second dimension takes on two possible values. Thus, a variable of type **RedCards** will have 26 (13 * 2) cells.

The first cell in the **RedHand** array will be **RedHand[2][diamond]**, the second will be **RedHand[2][heart]**. The last two elements in the array will be **RedHand[14][diamond]** and **RedHand[14][heart]** — for the Aces of diamonds and hearts, respectively. As you can see from this example, the different dimensions of an array don't have to be of the same type.

The **AnyCards** type definition is similar to that for **RedCards**. However, the second dimension for **AnyCards** spans the entire range of values that **CardSuit** can take on. This is allowed since, strictly, speaking, the range of possible value is itself a subrange.

Although the compiler lets you define a subrange type, it only catches *explicit* attempts to assign values beyond that range. For example, if you try to assign the value 11 to a variable of type **SingleDigit** directly, the compiler will complain that the constant is out of range.

On the other hand, if you first assign the 11 to an **INTEGER** variable, and then assign this value to a **SingleDigit** variable, you won't get any error messages from the compiler. The program will crash at run-time, however.

The RECORD Type

Sometimes you'll want to group items of information that are not of the same type. For example, suppose you wanted to represent information about a country. You might want to know the country's name, its capital, population, and area. The name and capital will be strings; the population and area will be whole numbers.

You could represent each of these items as a separate variable. However, that would quickly get tedious and confusing, especially if you have multiple countries.

Pascal's **RECORD** type is ideal for creating such a variable. A record represents information containing multiple components, which need not be of the same type.

The following listing shows a **TYPE** definition for a variable to represent a country.

```
TYPE Country = RECORD
                 Name, Capital : STRING[80];
                 Pop, Area : LONGINT;
               END; (* RECORD Definition *)
```

This definition describes a data type with four components: two 80-character strings and two **LONGINT**s. The components are specified between the reserved word **RECORD** and the **END** that terminates the **RECORD** definition.

Within these bounds, the components are listed in the same format as for variable declarations: identifiers, followed by a type (specified after a colon). These components are known as *fields* of the record.

A record definition is one of three places in Pascal where you'll find an **END** without a corresponding **BEGIN**. (This also occurs in a **CASE** statement and in an object definition.)

Accessing Fields in a Record

The following program shows how to access the individual fields of a **Country**.

Listing 7.7

```
(* Program to illustrate use of RECORD variables *)

PROGRAM RecordDemo;

TYPE Country = RECORD
                 Name, Capital : STRING[80];
                 Pop, Area : LONGINT;
               END; (* RECORD Definition *)
VAR Sample : Country;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Assign the specified values to the appropriate
     fields of TheCountry.
  *)
  PROCEDURE InitCountry ( VAR TheCountry : Country;
                          CName, Capital : STRING;
                          Population, Size : LONGINT);
  BEGIN
    (* To specify a field:
       <variable name> "." <field name>
    *)
    TheCountry.Name := CName;
    TheCountry.Capital := Capital;
    TheCountry.Pop := Population;
    TheCountry.Area := Size;
  END; (* InitCountry *)

  (* Display the contents of the individual fields of
     TheCountry.
  *)
  PROCEDURE DispCountry ( TheCountry : Country);
  BEGIN
    WRITE ( 'NAME      : ', TheCountry.Name:20);
    WRITELN ( ';  CAPITAL: ', TheCountry.Capital:17);
    WRITE ( 'POPULATION: ', TheCountry.Pop:10, ' ':10);
    WRITELN ( ';  AREA   : ', TheCountry.Area:10, ' sq.mi.');
  END; (* DispCountry *)

BEGIN  (* Main program *)
  InitCountry ( Sample, 'Finland', 'Helsinki',
              4900000, 130119);
  DispCountry ( Sample);
  AwaitUser;
END.
```

This program defines a **Country** type, then declares **Sample** as a variable of this type. The two procedures, **InitCountry** and **DispCountry**, do the program's work.

Once a particular **RECORD** type has been defined, declaring variables of that type is the same as declaring any other type of variable.

Procedures **InitCountry** and **DispCountry** illustrate that you can pass records as parameters — either by value or by reference. The syntax for doing this is the same as for passing any other type of variable.

You can leave a space between the period and the identifiers on either side. Thus, the following statements are equivalent. The spacing has no effect.

```
TheCountry.Area := Size;
TheCountry . Area := Size;
TheCountry .Area := Size;
TheCountry. Area := Size;
```

When **DispCountry** is called, the **Sample** record is copied, and the copy is stored as **TheCountry**. The procedure produces the following output:

```
NAME      :               Finland;  CAPITAL:          Helsinki
POPULATION:    4900000          ;  AREA   :     130119 sq.mi.
```

Structured Types As RECORD *Fields*

The **Country** record has two **STRING** fields. Recall that a **STRING** is an array of characters. As this indicates, you can have aggregate types as fields of a record.

In fact, you can have any type as a record field — except for a file (discussed in Chapter 8) and another record of the same type. The latter restriction says that you can't have any fields of type **Country** in the **Country** record. (You can have something called a *pointer* to a **Country** as a field, however. This leads to some very powerful data structures. Pointers are discussed in Chapter 9.)

Thus, you can have an array of integers or even an array of strings as fields. You can even have a record as a field for another record. The following listing shows examples of such record types, and includes variable declarations for the record types.

```
TYPE FreqArray = ARRAY [ 0 .. 10] OF INTEGER;
     CoinTrials = RECORD
                    Freqs : FreqArray;
                    Probability : REAL;
                  END; (* CoinTrials RECORD *)

     Menu = ARRAY [ 1 .. 20] OF STRING;
     MenuRec = RECORD
                 Choices : Menu;
                 ShowExtendedMenu : BOOLEAN;
               END; (* MenuRec RECORD *)

     MiscInfo = RECORD
```

```
                  Money : STRING[80];
                  Density, Literacy : REAL;
                END; (* MiscInfo RECORD *)
     Country = RECORD
                 Name, Capital : STRING[80];
                 Pop, Area : LONGINT;
               END; (* Country RECORD *)
     Nation = RECORD
                Land : Country;
                Data : MiscInfo;
              END; (* Nation RECORD *)

VAR Flips : CoinTrials;
    CurrMenu : MenuRec;
    SampleNation : Nation;
```

The **CoinTrials** structure has two fields. One of these fields is an array of type **FreqArray**, which contains 11 integers. The following procedure shows how to access each of the elements in **Flips**:

```
PROCEDURE InitCoinTrials ( VAR Vals : CoinTrials);
VAR Index : INTEGER;
BEGIN
  Vals.Probability := 0.5;
  (* To access the array: Vals.Freqs;
     to access a cell in the array: Vals.Freqs [ index]
  *)
  FOR Index := 0 TO 10 DO
    Vals.Freqs [ Index] := 0;
END; (* InitCoinTrials *)
```

The **MenuRec** structure also has two fields. However, this time one of them is a two-dimensional array. The access rules are a combination of the rules for accessing **Freqs** in a **CoinTrials** record and the rules for accessing a cell in a two-dimensional array.

To access a particular character in a string, first access the string, then specify the index for the character. For instance:

```
CurrMenu.Choices [ 3][ 4];
```

specifies the fourth character in the third string in the **Choices** field of **CurrMenu**.

To determine how to access fields when these fields are records, you just need to extend the rules for accessing first-level fields. To access a field in a record variable, you specify the variable's name, and the field name, separated by a period.

If the field happens to be a structured type, you may need to provide more information in order to get to the element you need. For example, you would use the following code to initialize **SampleNation**:

Listing 7.8

```
(* Program to illustrate how to access record fields that are
   also structured types.
*)

PROGRAM MoreRecDemo;

TYPE MiscInfo = RECORD
                  Money : STRING[80];
                  Density, Literacy : REAL;
                END; (* MiscInfo RECORD *)

     Country = RECORD
                 Name, Capital : STRING[80];
                 Pop, Area : LONGINT;
               END; (* Country RECORD *)
     Nation = RECORD
                Land : Country;
                Data : MiscInfo;
              END; (* Nation RECORD *)

VAR SampleNation : Nation;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
  WRITE ( 'Press ENTER to continue.');
  READLN ( Ch);
END; (* AwaitUser *)

(* Assign the specified values to the appropriate
   fields of TheCountry.
*)
PROCEDURE InitCountry ( VAR TheCountry : Country;
                        CName, Capital : STRING;
                        Population, Size : LONGINT);
BEGIN
  (* To specify a field:
     <variable name> "." <field name>
  *)
  TheCountry.Name := CName;
  TheCountry.Capital := Capital;
  TheCountry.Pop := Population;
  TheCountry.Area := Size;
END; (* InitCountry *)

(* Display the contents of the individual fields of
   TheCountry.
*)
PROCEDURE DispCountry ( TheCountry : Country);
BEGIN
  WRITE ( 'NAME      : ', TheCountry.Name:20);
  WRITELN ( ';  CAPITAL: ', TheCountry.Capital:17);
  WRITE ( 'POPULATION: ', TheCountry.Pop:10, ' ':10);
  WRITELN ( ';  AREA   : ', TheCountry.Area:10, ' sq.mi.');
END; (* DispCountry *)

(* Display the contents of the individual fields of
   TheNation.
*)
PROCEDURE DispNation ( TheNation : Nation);
BEGIN
  DispCountry ( TheNation.Land);
  WRITELN ( 'Currency  = ', TheNation.Data.Money);
  WRITELN ( 'Density   = ', TheNation.Data.Density : 10:2,
            ' persons per sq mi');
```

```
    WRITELN ( 'Literacy  = ', TheNation.Data.Literacy : 10:2,
              '%');
  END; (* DispNation *)

BEGIN  (* Main program *)
  InitCountry ( SampleNation.Land, 'Finland', 'Helsinki',
                4900000, 130119);
  SampleNation.Data.Money := 'Markka';
  SampleNation.Data.Density := 37.7;
  SampleNation.Data.Literacy := 100;
  DispNation ( SampleNation);
  AwaitUser;
END.
```

Procedure **InitCountry** is from an earlier program. The call to this procedure shows how to pass a record as a parameter — even if this record is a field in a larger record. Thus, to initialize the components of **SampleNation**'s **Land** field, just call **InitCountry** with this field as the first argument.

Within the procedure, it makes no difference whether the record argument is stand-alone or part of another data structure. The local name is used to access components, so the true variable's full name never becomes an issue.

The rest of the program body shows how to access a field within a field. First, access the **Data** field of **SampleNation** — that is, specify **SampleNation.Data**. This is just the name of a record. To access a field within this record, include a period and then the field name:

```
SampleNation.Data.Literacy
```

Using *WITH* to Shorten Field Names

You can have records as fields of other records to an arbitrarily deep level. To specify such deep fields, you simply extend the naming process used in the preceding example. The identifier up to the rightmost period must be a record name, and the identifier following the period must be a field. Beyond that, your name can have an arbitrary number of periods — each indicating a record at another level.

If you continue this process, you can end up with extremely long names, such as in the following listing:

Listing 7.9

```
(* Program to illustrate long field names *)

PROGRAM LongFieldNameDemo;

TYPE ShortStr = STRING[30];

     Education = RECORD
                   Major, Degree : ShortStr;
                   GPA : REAL;
                 END; (* Education RECORD *)

     Student = RECORD
                 Schooling : Education;
                 ID : ShortStr;
                 ClassOf : INTEGER;
               END; (* Student RECORD *)

     Graduate = RECORD
                  FirstName, LastName : ShortStr;
                  Age : INTEGER;
                  Learning : Student;
                END; (* Graduate RECORD *)

VAR Grad : Graduate;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)
```

```
 (* Prompt user and read an INTEGER. *)
PROCEDURE GetInteger ( Message : STRING;
                       VAR Value : INTEGER);
VAR TempStr : STRING;
BEGIN
  WRITE ( Message, ' ');
  READLN ( Value);
END; (* GetInteger *)

PROCEDURE GetShortStr ( Message : STRING;
                        VAR Value : ShortStr);
BEGIN
  Write ( Message, ' ');
  READLN ( Value);
END; (* GetShortStr *)

PROCEDURE GetReal ( Message : STRING;
                    VAR Value : REAL);
BEGIN
  Write ( Message, ' ');
  READLN ( Value);
END; (* GetReal *)

PROCEDURE GetGraduateInfo ( VAR Info : Graduate);
BEGIN
  GetShortStr ( 'First name?', Info.FirstName);
  GetShortStr ( 'Last name?', Info.LastName);
  GetInteger ( 'Age?', Info.Age);
   (* Get values for Learning fields *)
  GetShortStr ( 'ID?', Info.Learning.Id);
  GetInteger ( 'Class of?', Info.Learning.ClassOf);
   (* Get Info for Learning.Schooling Fields *)
  GetShortStr ( 'Major?', Info.Learning.Schooling.Major);
  GetShortStr ( 'Degree?', Info.Learning.Schooling.Degree);
  GetReal ( 'GPA?', Info.Learning.Schooling.GPA);
END; (* GetGraduateInfo *)

PROCEDURE DispGraduateInfo ( Info : Graduate);
BEGIN
```

```
    WRITELN ( 'First name: ', Info.FirstName);
    WRITELN ( 'Last name : ', Info.LastName);
    WRITELN ( 'Age       : ', Info.Age);
    (* Display values for Learning fields *)
    WRITELN ( 'ID        : ', Info.Learning.Id);
    WRITELN ( 'Class of  : ', Info.Learning.ClassOf);
    (* Display values for Learning.Schooling fields *)
    WRITELN ( 'Major     : ', Info.Learning.Schooling.Major);
    WRITELN ( 'Degree    : ', Info.Learning.Schooling.Degree);
    WRITELN ( 'GPA       : ', Info.Learning.Schooling.GPA:5:2);
  END; (* DispGraduateInfo *)

BEGIN
  GetGraduateInfo ( Grad);
  DispGraduateInfo ( Grad);
  AwaitUser;
END.
```

The first two records defined (**Education** and **Student**) are used as types for fields in other records. Thus, the field **Schooling** (in **Student**) is of type **Education**. In turn, the field **Learning** (in **Graduate**) is of type **Student**.

As you can see from **GetGraduateInfo** and **DispGraduateInfo**, it can get quite tedious to write out such a full name each time.

Pascal's **WITH** statement helps you avoid such lengthy names. The following versions of the two procedures from **LongFieldNameDemo** show how:

```
 (* Get values for Graduate fields.
    Procedure shows how to use WITH statements to
    shorten field names.
 *)
 PROCEDURE WGetGraduateInfo ( VAR Info : Graduate);
 BEGIN
   WITH Info DO
   BEGIN
     GetShortStr ( 'First name?', FirstName);
     GetShortStr ( 'Last name?', LastName);
     GetInteger ( 'Age?', Age);
```

```
        (* Get values for Learning fields *)
        WITH Learning DO
        BEGIN
          GetShortStr ( 'ID?', Id);
          GetInteger ( 'Class of?', ClassOf);
          (* Get Info for Learning.Schooling Fields *)
          WITH Schooling DO
          BEGIN
            GetShortStr ( 'Major?', Major);
            GetShortStr ( 'Degree?', Degree);
            GetReal ( 'GPA?', GPA);
          END; (* WITH (Info.Learning.)Schooling *)
        END; (* WITH (Info.)Learning *)
      END; (* WITH Info *)
    END; (* WGetGraduateInfo *)

    (* Display contents of a Graduate Record.
       Procedure also uses WITH statements to shorten field
names.
    *)
    PROCEDURE WDispGraduateInfo ( Info : Graduate);
    BEGIN
      WITH Info DO
      BEGIN
        WRITELN ( 'First name: ', FirstName);
        WRITELN ( 'Last name : ', LastName);
        WRITELN ( 'Age       : ', Info.Age);
      END; (* WITH Info *)
      (* Display values for Learning fields *)
      WITH Info.Learning DO
      BEGIN
        WRITELN ( 'ID        : ', Id);
        WRITELN ( 'Class of  : ', ClassOf);
      END; (* WITH Info.Learning *)
      (* Display values for Learning.Schooling fields *)
      WITH Info.Learning.Schooling DO
      BEGIN
        WRITELN ( 'Major     : ', Major);
        WRITELN ( 'Degree    : ', Degree);
        WRITELN ( 'GPA       : ', GPA:5:2);
      END; (* WITH (Info.Learning.)Schooling *)
    END; (* WDispGraduateInfo *)
```

Independent **WITH** Statements

The **DispGraduateInfo** routine contains three independent **WITH** statements. The first one specifies that the record variable **Info** is meant whenever certain field names are used. This holds for the compound statement following the **WITH Info** specification. Thus, the

FirstName in the procedure refers to **Info.FirstName**; similarly, **Age** refers to **Info.Age**.

If you specify **FirstName** in a statement that is not part of any **WITH** statement, the compiler would look only at ordinary variables. The compiler would not look at components of structured variables.

Because of the **WITH** statement, however, the compiler first checks whether **Info** has a field named **FirstName**. If the record has no such field, the compiler looks through its local environment for a variable with this name.

Notice that **Info.Age** is used, instead of just **Age**. This was done deliberately to show that you can still use the full name for a record field, even within a **WITH** statement. In the example, both **Info.Age** and **Age** refer to the same memory location. It's not good programming practice to use two different names for the same variable in the same context. This was done here merely to illustrate a point.

The second **WITH** statement also specifies a record. This one is a nested record, however. The full name of the record is **Info.Learning**. In this case, the compiler first checks this variable for a field named **Id**. If it doesn't find one, then it checks the local environment.

Finally, the third **WITH** statement specifies an even more deeply nested record, namely, **Info.Learning.Schooling**. Because the three **WITH** statements are independent of each other, the shortened versions of the names apply only in the specified parts of the procedure. For example, you could not refer to **Id** inside the third **WITH** statement.

Nested **WITH** Statements

The **WITH** statements in **GetGraduateInfo** are nested. This means that you can use shortened names when specifying records in the inner **WITH** statements. You can specify **Learning** as the record to use because **Info** is assumed, and you can specify **Schooling** because **Info.Learning** is assumed.

Nesting the **WITH** statements also changes the scope of the first two **WITH**s. The first **WITH** is within scope throughout the procedure — as you can see by the **END** that terminates its compound statement.

Using Names for Variables and for Fields

The following version of **DispGraduateInfo** is identical to the previous one, except that two lines have been added. The local variable, **ClassOf** has been declared in the procedure, and has been assigned the value 1538 outside the scope of any **WITH** statements.

```
(* Display contents of a Graduate Record.
   Procedure uses WITH statements to shorten field names.
*)
```

```
PROCEDURE WDispGraduateInfo ( Info : Graduate);
VAR ClassOf : INTEGER;
ClassOf := 1538;
BEGIN
  WITH Info DO
  BEGIN
    WRITELN ( 'First name: ', FirstName);
    WRITELN ( 'Last name : ', LastName);
    WRITELN ( 'Age       : ', Info.Age);
  END; (* WITH Info *)
  (* Display values for Learning fields *)
  WITH Info.Learning DO
  BEGIN
    WRITELN ( 'ID        : ', Id);
    WRITELN ( 'Class of  : ', ClassOf);
  END; (* WITH Info.Learning *)
  (* Display values for Learning.Schooling fields *)
  WITH Info.Learning.Schooling DO
  BEGIN
    WRITELN ( 'Major     : ', Major);
    WRITELN ( 'Degree    : ', Degree);
    WRITELN ( 'GPA       : ', GPA:5:2);
  END; (* WITH (Info.Learning.)Schooling *)
END; (* WDispGraduateInfo *)
```

What value do you think will be displayed when

```
WRITELN ( 'Class of  : ', ClassOf);
```

is called? If you run the program **LongFieldNameDemo** with the modified procedure, you'll find that the value you entered for **Info.Learning.ClassOf** is displayed. This is because the call to **WRITELN** occurs within the scope of the second **WITH** statement. As a result, the record field **(Info.Learning.)ClassOf** has greater visibility. (To avoid confusion, don't duplicate variable names in the first place.)

Arrays of Records

You've seen that you can have records that contain arrays as fields. Earlier, you learned that the base type of an array could be just about any type. Can you have a record as the base type for an array?

The following listing shows that you can define an array of records, and also shows how to access individual cells of the array, and individual fields of a record (that is, of a cell).

Listing 7.10

```
(*  Program to illustrate definition and use of
                  Arrays of records.
              *)

PROGRAM RecordTest;
USES Windows;

CONST MaxVal = 26;
      MaxTrials = 1000;

TYPE Entry = RECORD
               Freq, Max, Min : INTEGER;
             END; (* Entry RECORD *)
     (* Each cell is an Entry RECORD *)
     IntArray = ARRAY [ 0 .. MaxVal] OF Entry;

VAR Data : IntArray;
    RawVal, Result, Count, RandSeed :INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

(* Initialize each cell of array to its appropriate values *)
PROCEDURE InitIntArray ( VAR Ints : IntArray);
VAR Index : INTEGER;
BEGIN
  FOR Index := 0 TO MaxVal DO
  BEGIN
    Ints [ Index].Freq := 0;
    Ints [ Index].Max := 0;
    Ints [ Index].Min := MAXINT;
  END; (* FOR Index := 0 TO MaxVal *)
END;  (* InitIntArray *)

(* Display the contents of each cell's fields *)
PROCEDURE DispIntArray ( VAR Ints : IntArray);
VAR Index : INTEGER;
BEGIN
  FOR Index := 0 TO MaxVal DO
  BEGIN
    WITH Ints [ Index] DO
      WRITE (Index:2, ':', Freq:3,
             ' (', Min:4, ')(', Max:5, ')  ');
      IF (Index MOD 3 = 2) THEN
        WRITELN;
  END;
  WRITELN;
END;  (* DispIntArray *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END;  (* GetInt *)

BEGIN (* Main *)
  GetInt ( 'Integer for random number generator', RandSeed);
  SRAND ( RandSeed);

  InitIntArray ( Data);
  (* Fill array cells with random values *)
  FOR Count := 1 TO MaxTrials DO
  BEGIN
    RawVal := RAND;
    Result := RawVal MOD ( MaxVal + 1);
    (* increment Freq field of cell whose
       index corresponds to Result
    *)
    INC ( Data [ Result].Freq);
    (* update Min and Max fields of same cell *)
    IF RawVal > Data [ Result].Max THEN
      Data [ Result].Max := RawVal;
    IF RawVal < Data [ Result].Min THEN
      Data [ Result].Min := RawVal;
  END;  (* FOR Count := 1 TO MaxTrials *)
  DispIntArray ( Data);
  AwaitUser;
END.
```

The main data structure for this program is a 27-element array. The individual cells of this array are actually records, each of which has three **INTEGER** fields.

For a sample run, this program produced the following output:

```
 0: 35 ( 162)(31698)   1: 24 (2053)(30025)   2: 35 ( 677)(32672)
 3: 40 (1029)(32538)   4: 34 ( 436)(31837)   5: 36 ( 545)(32648)
 6: 39 (1734)(32028)   7: 34 (  34)(32677)   8: 44 (  35)(30599)
 9: 35 (2493)(31437)  10: 37 ( 712)(32194)  11: 32 (1820)(30791)
12: 26 (1524)(32250)  13: 43 ( 364)(32521)  14: 38 ( 203)(30929)
15: 37 ( 663)(32658)  16: 42 (1204)(32659)  17: 39 (1232)(32498)
18: 38 (1260)(31284)  19: 56 (1612)(32527)  20: 42 ( 236)(32690)
21: 34 ( 453)(32610)  22: 36 (1075)(31936)  23: 30 ( 374)(29966)
24: 44 (  51)(32586)  25: 42 ( 241)(31264)  26: 28 (3104)(31130)
```

To produce such a run, this program:

- Generates random integers and determines the remainder when divided by 27.
- Updates the array cell whose index corresponds to the remainder.

In particular, the program increases the frequency count in the **Freq** field of the appropriate array cell. The program also checks (and updates, if necessary) the **Min** and **Max** fields, which contain the smallest and largest values that have produced the given remainder.

Thus, in the sample run, 35 of the 1000 values left a remainder of 0 when divided by 27. The smallest of these values was 162, and the largest was 31698.

Accessing Records in the Array

To determine how to access a particular cell, just work through the task step by step. First, try to specify a record. In this case, the record is a cell of an array, so the first step is to specify an array cell.

To specify an array cell, supply the variable name, followed by the cell's index within square brackets. For example,

```
Data [ Result]
```

specifies the record whose index is the current value of **Result**. This object is a record.

cOnce you've specified the record, you need to specify the field you want. To do this, add a period and then the field identifier. For example,

```
Data [ Result].Max
```

specifies the **Max** field of the record with index value **Result**.

The **WITH** Statement Revisited

Remember, the **WITH** statement enables you to shorten record names. You can do this even if the record happens to be an array cell, as shown in **DispIntArray** in the previous program. However, be careful of how and where you specify the **WITH** variable under certain conditions. For example, consider the following version of **DispIntArray**:

```
(* INCORRECT routine to
   display the contents of each cell's fields.
   Routine is incorrect because of WITH statement location.
*)
PROCEDURE DispIntArray ( VAR Ints : IntArray);
VAR Index : INTEGER;
BEGIN
  Index := 0;
  (* This will always refer to the 0th cell *)
  WITH Ints [ Index] DO
  BEGIN
    WHILE Index <= MaxVal DO
    BEGIN
      WRITE (Index:2, ':', Freq:3,
             ' (', Min:4, ')(', Max:5, ')  ');
      IF (Index MOD 3 = 2) THEN
        WRITELN;
      INC ( Index);
    END; (* WHILE Index <= MaxVal *)
  END;
  WRITELN;
END;  (* DispIntArray *)
```

This procedure uses a **WHILE** loop to go through the array. There is also a **WITH** statement; however, in this version, the loop is inside the scope of the **WITH** statement.

To determine what cell is meant, **Index** must be evaluated. When the procedure reaches the **WITH** statement, **Index** evaluates to 0. Once inside the **WHILE** loop, the value of **Index** is never used again to specify a new cell. As a result, the procedure always displays **Ints[0]**.

Operations on Records

You can assign the contents of an entire record to a variable of exactly the same type. This assignment requires only one statement. To assign an entire record, an area of memory is copied at another location. The fact that the two variables are of the same type assures that the memory areas will be the same size. Besides this total assignment, the only other operations on records must be carried out on a component to component basis.

Variant Records

Suppose you need to keep information about individual persons who range from children through married adults. Much of the information you store will be the same for all these age groups. For example, you may want to store first and last name, age, and sex for each person.

However, there will also be a few items that are pertinent for one age group but not for others. For example, you may want income and marital status data for adults and possibly teenagers. Similarly, you might want to know whether a child is in preschool or school, and you may want to know a child's grade and school performance.

Pascal allows you to create a type definition for a *variant record*. In each instance of such a record, you can store whichever collection of special information is relevant for that instance. The following listing shows such a variant record:

Listing 7.11

```
(* Program to illustrate storage requirements of
   variant records, with tag field.
*)

PROGRAM VarRecDemo;

TYPE Sex = ( female, male);
     MaritalStatus = ( single, married, divorced, widowed);
     AgeGroup = ( infant, toddler, child, teenager, adult);
     Person = RECORD
                First, Last : STRING;
                Age : INTEGER;
                Gender : Sex;
                Weight, Height : REAL;
                (* Variant part begins here;
                   variant used depends on value of WhichGroup.
                *)
                CASE WhichGroup : AgeGroup OF
                  (* if WhichGroup = infant, then
                     record has two REAL fields.
                  *)
                  infant : ( BirthWeight, BirthHeight : REAL);
                  toddler : ( InPreSchool, InDayCare : BOOLEAN);
                  child : ( Grade : INTEGER);
                  (* If WhichGroup = teenager, then
                     record has one INTEGER and two REAL fields.
                     Note that tGrade is used here, since Grade
                     has already been used, and cannot be used
                     for a field name twice. The same holds for
                   tIncome.
                *)
                teenager : ( GPA, tIncome : REAL;
                             tGrade : INTEGER);

                adult : ( Income : REAL;
                          Status : MaritalStatus);
            END; (* Person RECORD *)

VAR Nemo : Person;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  WRITELN ( 'Nemo requires ', SizeOf ( Nemo), ' bytes.');
  Nemo.Income := 39000.50;
  WRITELN ( Nemo.Income: 10: 2);
  AwaitUser;
END.
```

Although this definition looks formidable, it's actually quite straightforward. The **Person** record has a fixed and a variant part. The syntax for the fixed part (fields **First** through **Height**) is the same as in record examples you've already seen.

The variant part begins with the **CASE WhichGroup** line. A particular **Person** record can store any of five different groups of fields — depending on the value of the record's **WhichGroup** field.

For example, if **Nemo.WhichGroup** is **teenager**, then the **Nemo** record is storing three items of variant information: **tGrade**, GPA, and **tIncome**. On the other hand, if **Nemo.WhichGroup** is **adult**, the record is storing **Status** and **Income** information. Since the value of **Nemo.WhichGroup** can, in principle, change during program execution, the program must be prepared to store any of the variant configurations.

To be able to store any of these configurations, a **Person** record must have enough room for the largest variant — just in case. In the example, this would be 10 bytes needed to store the fields for a **teenager**.

If you run the **VarRecDemo** program, you'll learn that **Nemo** takes 540 bytes: 256 bytes for each of the strings, 2 bytes for **Age**, 4 bytes each for **Height** and **Weight**, 1 byte each for **Gender** and **WhichGroup**, 10 bytes for the **teenager** variant configuration, and 6 bytes for overhead to handle a variant record.

At any given time, only one of the field configurations is the correct one, but you can access any configuration whenever you wish. (Of course, you may get garbage if you access for a teenager when the data stored are for a toddler.)

To specify the fields for a particular configuration, you need to list them within parentheses. Within the parentheses, the list will have the same format as a variable declaration.

The variant part of a record definition must follow the fixed part. The variant part always begins with **CASE**. The record definition ends with a single **END**, whether it's a variant or a fixed record.

Tagged and Untagged Variants

The **WhichGroup** field in the preceding definition is known as a *tag field*. This field acts as a filter — to enable you to check which of the variants to choose. Your program checks the value of this field in order to determine which variant configuration to read or write. A variant record such as **Person** is a *tagged variant* because it has a tag field.

In the preceding program, notice that you don't *have* to do anything with the **WhichGroup** field in order to use a specific variant. While it's not a syntax error to access variants without going through **WhichGroup**, it can be dangerous, since your program will always need to remember the configuration that is currently being used. Otherwise, the program could access garbage values by using the wrong configuration.

There is another way of specifying a record variant. You can omit the tag field, and just specify the type whose values are used to specify the variants. The following listing shows a record definition in which the tag is omitted. **ZPerson** is identical to **Person** in the preceding program, except that the **CASE** line is different.

Listing 7.12

```
(* Program to illustrate storage requirements of
   variant records, without tag field.
*)
```

```
PROGRAM VarRecDemo2;

TYPE Sex = ( female, male);
     MaritalStatus = ( single, married, divorced, widowed);
     AgeGroup = ( infant, toddler, child, teenager, adult);
     ZPerson = RECORD
                First, Last : STRING;
                Age : INTEGER;
                Gender : Sex;
                Weight, Height : REAL;
                (* Variant part begins here;
                   variant used depends on AgeGroup values
                *)
                CASE AgeGroup OF
                  infant : ( BirthWeight, BirthHeight : REAL);
                  toddler : ( InPreSchool, InDayCare : BOOLEAN);
                  child : ( Grade : INTEGER);
                  teenager : ( GPA, tIncome : REAL;
                               tGrade : INTEGER);

                  adult : ( Income : REAL;
                            Status : MaritalStatus);
              END; (* Person RECORD *)

VAR ZNemo : ZPerson;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  WRITELN ( 'ZNemo requires ', SizeOf ( ZNemo), ' bytes.');
  ZNemo.Income := 39000.50;
  WRITELN ( ZNemo.Income: 10: 2);
  AwaitUser;
END.
```

By omitting the tag, you also delete the **WhichGroup** field from the record definition. When you run the preceding program, you'll learn that **ZNemo** takes 536 bytes — four fewer bytes than **Nemo**. This is because **ZNemo** doesn't have a **WhichGroup** field and because the untagged variant requires less overhead.

Using an untagged variant is equivalent to bypassing the **WhichGroup** field in the tagged version. It saves you a bit of storage, and can save you the accesses needed to update and check the tag field. However, you lose the ability to check which variant is currently valid. If you need to use variant records, you're generally better off using tagged variants.

The OBJECT Type: A Preview

With CodeWarrior Pascal, you can do *object oriented programming* (OOP). This is a very powerful programming technique, and will be discussed in Chapters 11 and 12. An important concept in OOP is that of an object. This section provides a brief preview of how objects are represented in CodeWarrior Pascal.

An *object* includes a collection of features that characterize the object. (It may also include other types of components, which will be discussed in Chapter 11.) For example, a sphere might be characterized as an object that has a particular size, or radius, and that is either solid or hollow.

Once you have a description of an object, you can define new objects that have the same features as the prototype object as well as some additional characteristics. For example, you could describe a beachball as something with both the features of a sphere and an additional feature — namely, a design on the surface. Or you could describe a sportsball as a sphere that is either bouncy or not.

In CodeWarrior Pascal, an object is represented in a manner very similar to a record. For example, the following would describe a **Sphere** as an object with two features:

```
TYPE Sphere = OBJECT
                radius : REAL;
                solid : BOOLEAN;
              END;  (* OBJECT definition *)
```

Syntactically, the only difference between this and a record definition is the word **OBJECT** instead of **RECORD**. This difference has important consequences, however, because you can define new objects that inherit the features of a **Sphere**. You could not do this if **Sphere** had been defined as a record.

OBJECT is a reserved word that is new in CodeWarrior Pascal, and that is used to describe a new data type. Once you have such a type definition, you can declare variables of the new type. Such variables are known as *instances* of the object. The following listing shows how to declare an instance of a **Sphere**.

```
EuclideanSphere : Sphere;
```

Once you've defined an object, you can define variants on it. These variants will *inherit* the object's features, and will also have unique features of their own. For example, the following listing shows how to define objects such as a **BeachBall** and a **SportsBall**, once a **Sphere** has been defined.

```
TYPE Sphere = OBJECT
                radius : REAL;
                solid : BOOLEAN;
              END;  (* OBJECT definition *)

     Design = ( solidcolor, patterns, pictures);

     BeachBall = OBJECT (Sphere)
                   Surface : Design;
                 END; (* OBJECT definition *)

     SportsBall = OBJECT ( Sphere)
                    Bouncy : BOOLEAN;
                  END; (* OBJECT definition *)

VAR MyBeachBall : BeachBall;
    Squashball, Basketball : SportsBall;
```

Objects that inherit features of another object are called *descendant objects*. In the example, **BeachBall** and **SportsBall** are descendant objects. Both have **Sphere** as their *ancestor object*. Descendant objects can have their own descendants.

To describe a descendant object, use an **OBJECT** definition. Next to the reserved word

OBJECT, however, you need to specify the ancestor object within parentheses. This allows the new object to inherit the ancestor object's features. Thus, it makes available all the fields of the ancestor object — just as if they had been included explicitly in the definition of the descendant object.

The syntax for accessing an object's fields is the same as for accessing a record's fields. For example, all of the following are valid statements:

```
Squashball.Solid := FALSE;
Squashball.Bouncy := TRUE;
MyBeachball.Radius := 12;
MyBeachball.Color := pictures;
```

You'll learn how to use objects in Chapter 11. There, you'll also find a more elegant and safer way to access an object's components. This will involve methods (procedures or functions) associated with the object.

The SET Type

Sometimes it's useful to keep collections of values for comparison. For example, you might want to check whether a particular character descends below the line when printed — that is, whether the character is *g* or *j* or *p* or *q* or *y*.

Or, suppose you're keeping track of students' majors. In addition to determining the number of people in each major, you might count the number of students with science or humanities majors. To do this for a science major you might check, for instance, whether a student's major is math, physics, chemistry, biology, biochemistry, astronomy, or earth science.

In such situations, you not only need to work with individual values, you also need to work with collections of values. Depending on the example, such collections may contain arbitrary combinations of values of the specified type. When you're working with such a collection, you're generally interested in whether or not a particular value is present. The actual value not be of direct interest.

In mathematics, such a collection of values, or elements, is called a set. A *set* is an unordered collection of elements in which repetitions are discarded. That is, an element appears at most once in a set. You can describe a set by listing the elements in the set. It doesn't matter how often you list an element; nor does the order matter.

For example, suppose you had the string “sweeteners," and you wanted to represent its constituent letters in a set. This set would contain six elements: *s*, w, *e*, t, *n*, and *r*. Notice that the `e' and `s' each appear only once in the set — despite the fact that these letters are repeated in the word.

Pascal's **SET** type enables you to represent such collections. The following program shows how to define and initialize sets. You won't see anything when you run this program. Later, you may want to add instructions to display the contents of the sets. As with the other structured types, you first need to describe the **SET** type you wish to use, as shown in the example program.

Listing 7.13

```
(* Program to illustrate use of SETs *)

PROGRAM SetDemo1;

     (* Define a type to represent a collection of characters *)
TYPE CharSet = SET OF CHAR;

     (* define a type whose values are majors *)
     Major = ( anthropology, art, astronomy, biochemistry,
               biology, chemistry, complit, earthsci,
               economics, english, literature, math, physics,
               politics, psychology);
     (* Define a type to represent a collection of majors *)
     MajorGroup = SET OF Major;

VAR AllMajors,Science : MajorGroup;
    WdSet : CharSet;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  (* Assign the entire collection of values to AllMajors *)
  AllMajors := [ anthropology .. psychology];
  (* Assign a list of majors to the set Science *)
  Science := [ physics, astronomy .. chemistry, earthsci];
  WdSet := [ 'b', 'd', 'f', 'h', 'k', 'l', 't'];
  AwaitUser;
END.
```

There are restrictions on the base type, which we'll discuss in a bit.

Once you've specified this set template, you can declare variables of the specified type. In the example program, the first type definition describes a **CharSet** as a set whose elements will be of type **CHAR**.

The **MajorGroup** type definition specifies a set whose base type is an enumerated type. Once the possible values of this enumerated type are listed, you can define types containing such values. Thus, the **MajorGroup** definition is valid, as are the declarations for **Science** and **AllMajors**.

The main program shows how to assign a value to a set variable. To do this, you use an ordinary assignment operator. However, on the right hand side of the assignment statement you need to use the set constructor. The *set constructor* — **[]** — is used to make the element or elements it contains into a set.

Thus, the expression

```
[ anthropology .. psychology]
```

creates a set which contains all 15 possible values for a **Major**. To specify the contents of a set, you simply need to list the elements. Notice that you can use the double dots (..) to indicate a range of values.

The next assignment shows another way to specify a collection of values. In this case, a subrange of values is indicated by using double dots. In addition, two other majors are specified by listing them. Individual elements are separated by commas.

After this second assignment statement, **Science** will contain the following values: **astronomy**, biochemistry, **biology**, chemistry, **earthsci**, and **physics**.

The fact that **physics** is listed first does not have any significance. Remember, a set is an *unordered* collection of elements. All that matters is whether or not a particular value is an element. Thus, the following assignments will all produce exactly the same set.

```
(* all four assignments make Science the same set *)
Science := [ physics, astronomy .. chemistry, earthsci];
Science := [ astronomy .. chemistry, earthsci, physics];
Science := [ astronomy .. chemistry, physics, earthsci];
Science := [ astronomy, biology, biochemistry, chemistry,
             physics, earthsci];
```

Returning to the **SetDemo1** program, the third assignment simply builds a set containing several individual characters, and assigns this set to **WdSet**. Can you guess what the characters included in **WdSet** have in common?

WARNING! Be careful not to confuse the set constructor brackets with the brackets for specifying an array cell. The context in your program should make it clear which way the brackets are being used.

Set Features and Restrictions

The base type of a set must be an ordinal type, but not all ordinal types are allowed as a base type for a set. For reasons that have to do with the implementation of a set, the base type can have at most 2048 possible values. Furthermore, the ordinal values of the set's base type must be between 0 and 2047, inclusive.

This means, for example, that you can't have a **SET OF INTEGER**. This type has too many (more than 2048) possible values, and the ordinal positions of these values lie outside the range of 0 to 2047.

You can, however, have a set with a subrange type as its base type. For example, the following definitions create a set whose elements can be any value from 20 through 199.

Listing 7.14

```
PROGRAM SubRangeSetDemo;

TYPE MiddleNrs = 20 .. 199;
     MiddleSet = SET OF MiddleNrs;
```

```
VAR MidVals : MiddleSet;
    Ch : CHAR;

BEGIN
  MidVals := [ 20, 199, 36];
  WRITELN ( 'Set MidVals was assigned 3 elements.');
  WRITE ( 'Press ENTER to end program.');
  READLN ( Ch);
END.
```

In this program, the base type of the set is an ordinal value (**MiddleNrs**). This type has fewer than 2048 values, and the ordinal values (20 .. 199) associated with its values are within the 0 .. 2047 range. Therefore, **MiddleNrs** satisfy all the restrictions on set types.

TIP: The base type of a set must be an ordinal type, subject to two restrictions:

1. the base type can have at most 2048 possible values

2. the ordinal positions of these values must be between 0 and 2047.

Representation of Sets

A set is represented as an array of bits. Each bit corresponds to a possible value in the set. If the bit is 1, the value is an element in the set; if the bit is 0, the value is not an element.

This representation should make it clear why order makes no difference when you specify elements in a set, and also why repetitions have no effect. The order is irrelevant, since all that matters is whether a particular bit is 1 or 0. Repetitions are irrelevant because turning on a bit that is already on has no effect.

A Special Set

There is one set that has a very special status. This is the set that contains no elements. Such a set is known as the *null*, or *empty*, set. This set is similar to 0 in arithmetic — it is needed, but has no effect in many operations.

The null set is indicated by []. For example, the following program creates and declares a set type, and then initializes this variable to an empty set.

Listing 7.15

```
PROGRAM NullSetDemo;

TYPE CharSet = SET OF CHAR;

VAR WdSet : CharSet;
     Ch : CHAR;

BEGIN
  (* After this assignment, WdSet has been initialized,
     but does not contain any elements.
  *)
  WdSet := [];
  WRITELN ('WdSet was set to the null set.');
  WRITE ( 'Press ENTER to end program.');
  READLN ( Ch);
END.
```

You'll learn how to use the null set in expressions, and you'll also find out what happens if you combine the null set with other sets.

Set Operators

What kinds things can you do with sets? You can add, select, and remove elements, and you can compare sets to determine whether they contain the same elements. You can also check whether a particular value is an element of a set.

Checking Whether an Element Is in a Set

It's often very useful to be able to check whether a specific value is an element in a set. In fact, it's essential to be able to do this — to display the contents of a set.

Pascal's **IN** operator enables you to check whether a value is contained in a set. This binary operator takes two *different* types of values — an element of the set's base type and a set. The operator returns a **BOOLEAN**. If the element is contained in the set, then the operator returns **TRUE**; otherwise the operator returns **FALSE**.

The following listing shows how to use the **IN** operator:

Listing 7.16

```
(* Program to illustrate use of IN operator *)

PROGRAM SetInDemo;

     (* Define a type to represent a collection of characters *)
TYPE CharSet = SET OF CHAR;

VAR WdSet : CharSet;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Display any values between StartCh and EndCh, inclusive,
     that are elements in TheSet.
  *)
  PROCEDURE DispCharSet ( TheSet : CharSet;
                          StartCh, EndCh : CHAR);
  VAR IndexCh : CHAR;
      Count : INTEGER;
  BEGIN
    Count := 1;
    FOR IndexCh := StartCh TO EndCh DO
      (* Note that IN takes a CHAR and a CharSet
         as operands in this procedure
      *)
      IF IndexCh IN TheSet THEN
      BEGIN
        WRITE ( IndexCh:3);
        IF Count MOD 20 = 0 THEN
          WRITELN;
        INC ( Count);
      END;
  END; (* DispCharSet *)

BEGIN
  WdSet := [ 'A' .. 'z'];
  DispCharSet ( WdSet, 'A', 'z');
  WRITELN;
  AwaitUser;
END.
```

Procedure **DispCharSet** displays the contents of the specified **CharSet**. This procedure exemplifies a typical strategy for determining or displaying elements in a set:

You loop through the specified range of base type values.

- You check whether each value is an element of the set.

There is no way of reading the contents of a set directly because there is no relationship between a bit in the set's representation and the value the bit represents. Instead, you use the **IN** operator in such a procedure. For each value within the specified range (between *A* and *z* in the program), the procedure checks whether the value is an element of the specified set. If so, the procedure writes the element.

In addition to illustrating the **IN** operator, **DispCharSet** also shows how to pass a set parameter by value. To do this, simply specify the set's name as an argument. To declare a set parameter, simply provide an identifier and an appropriate type.

The program produces the following output:

```
A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T
U  V  W  X  Y  Z  [  \  ]  ^  _  `  a  b  c  d  e  f  g  h
i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
```

If you check an ASCII table, you'll find that these are all the characters between `A' and `z.'

The precedence for the **IN** operator is low — it's the same as for the comparison operators (=, <=, and so forth).

The Set Union Operator

To add to a set, you need to use the *set union operator*, also called the *set addition operator* (+). The following listing shows how to do this:

Listing 7.17

```
(* Program to illustrate use of set union operator *)

PROGRAM SetUnionDemo;

     (* Define a type to represent a collection of characters *)
TYPE CharSet = SET OF CHAR;

VAR WdSet1, WdSet2 : CharSet;
    IndexCh : CHAR;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Display any values between StartCh and EndCh, inclusive,
     that are elements in TheSet.
  *)
  PROCEDURE DispCharSet ( TheSet : CharSet;
                               StartCh, EndCh : CHAR);
  VAR IndexCh : CHAR;
      Count : INTEGER;
  BEGIN
    Count := 1;
    FOR IndexCh := StartCh TO EndCh DO
      (* Note that IN takes a CHAR and a CharSet
         as operands in this procedure
      *)
      IF IndexCh IN TheSet THEN
      BEGIN
        WRITE ( IndexCh:3);
        IF Count MOD 20 = 0 THEN
          WRITELN;
        INC ( Count);
      END;
  END; (* DispCharSet *)

BEGIN   (* Main program *)
  WdSet1 := [ 'b', 'd', 'f', 'h', 'k', 'l', 't'];
  (* Build WdSet by combining contents of two sets *)
  WdSet2 := WdSet1 + [ 'g', 'j', 'p', 'q', 'y'];
  WRITELN ( 'WdSet1:');
  DispCharSet ( WdSet1, 'a', 'z');
  WRITELN;
  WRITELN ( 'WdSet2:');
  DispCharSet ( WdSet2, 'a', 'z');
  WRITELN;
  AwaitUser;
END.
```

This program builds **WdSet2** by combining the contents of two other sets. The set union operator takes two sets as operands, and returns a third set as the result. This set is called the *union* of the two operand sets. The third (union) set contains all elements that are in either or both of the two operand sets.

Notice that no set constructor was used with **WdSet1** on the right hand side of the assignment in the main program. This is because **WdSet1** already is a set. If you wrote

```
[WdSet1]
```

you would create a set whose single element was itself a set. Such constructs are allowed in mathematics, but not in Pascal.

The **SetUnionDemo** program produces the following output:

```
WdSet1:
  b   d   f   h   k   l   t
WdSet2:
  b   d   f   g   h   j   k   l   p   q   t   y
```

Recall that repetitions are discarded in sets. This means that an element contained in both operand sets will only appear once in the resulting set. For example, the following assignment line does not create a different **WdSet2** than the program — despite the fact that the second and third sets have common elements.

```
WdSet2 := WdSet1 + ['g', 'j', 'p', 'q', 'y'] + ['g', 'y', 'y'];
```

The preceding statement also shows that you can list the same element twice in a set without causing a compiler error or repeating the element in the set. Listing an element twice has no effect.

TIP: The union (+) of two sets is a set that contains all elements that are contained in either or both of the component sets. If an element is contained in both component sets, it still appears only once in the union set. The union of two sets will always contain at least as many elements as the larger of the two operand sets.

Set Unions Involving the Null Set

What do you think the following program will display?

Listing 7.18

```
(* Program to illustrate unions involving null set *)

PROGRAM UnionDemo2;

TYPE CharSet = SET OF CHAR;

VAR WdSet : CharSet;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Display any values between StartCh and EndCh, inclusive,
      that are elements in TheSet.
  *)
  PROCEDURE DispCharSet ( TheSet : CharSet;
                          StartCh, EndCh : CHAR);
  VAR IndexCh : CHAR;
      Count : INTEGER;
  BEGIN
    Count := 1;
    FOR IndexCh := StartCh TO EndCh DO
      (* Note that IN takes a CHAR and a CharSet
         as operands in this procedure
      *)
      IF IndexCh IN TheSet THEN
      BEGIN
        WRITE ( IndexCh:3);
        IF Count MOD 20 = 0 THEN
          WRITELN;
        INC ( Count);
      END;
  END; (* DispCharSet *)

BEGIN
  WdSet := ['g' .. 'k', 'k', 'z'];
  WdSet := WdSet + [];
  DispCharSet ( WdSet, 'a', 'z');
  WRITELN;
  AwaitUser;
END.
```

The second assignment statement assigns the union of **WdSet** and another set to **WdSet**. The other set is the empty set, which contains no elements. The result of combining a set with no elements with another set will simply be the other set. Thus, the assignment will not affect the value of **WdSet**.

TIP: The union of any set with the null set is the original set. In this context, the null set behaves like 0 does in addition — it changes nothing.

The set union operator is essentially an additive operator, and has the same precedence as other additive operators. This precedence is below that for multiplicative operators but above that for comparison operators.

The Set Intersection Operator

Whereas the set union operator creates a set made up of elements combined from two sets, the *set intersection operator* (*) creates a set of elements *selected* from other sets.

This binary operator takes two set operands and returns a third set that contains only those elements that are contained in *both* of the operand sets. This resultant set is known as the *intersection* of the two operand sets and it has *at most* as many elements as the smaller operand set.

The following program shows how to use the set intersection operator:

Listing 7.19

```
(* Program to illustrate use of set intersection operator *)

PROGRAM IntersectionDemo;
USES Types;

CONST Smaller = 3;
      Larger = 4;
TYPE ByteSet = SET OF BYTE;
VAR BySmaller, ByLarger, Result : ByteSet;
    Count : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Display any values between StartByte and EndByte,
     inclusive, that are elements in TheSet.
  *)
  PROCEDURE DispByteSet ( TheSet : ByteSet;
                          StartByte, EndByte : BYTE);
  VAR IndexByte : BYTE;
      Count : INTEGER;
  BEGIN
    Count := 1;
    FOR IndexByte := StartByte TO EndByte DO
      (* Note that IN takes a BYTE and a ByteSet
         as operands in this procedure
      *)
      IF IndexByte IN TheSet THEN
      BEGIN
```

```
        WRITE ( IndexByte:4);
        IF Count MOD 15 = 0 THEN
          WRITELN;
        INC ( Count);
      END;
  END; (* DispByteSet *)

  (* Display contents of the specified set. *)
  PROCEDURE ShowSetData ( TheSet : ByteSet;
                          Message : STRING);
  BEGIN
    WRITELN ( Message, ':');
    DispByteSet ( TheSet, 0, 255);
    WRITELN;
  END;  (* Show SetData *)

  (* Add the appropriate elements to TheSet:
     20 values, separated by Increment.
  *)
  PROCEDURE BuildSet ( VAR TheSet : ByteSet;
                       Increment : INTEGER);
  VAR Count, Val : INTEGER;
  BEGIN
    Val := 0;
    FOR Count := 1 TO 20 DO
    BEGIN
      Val := Val + Increment;
      TheSet := TheSet + [ Val];
    END; (* FOR Count := 1 TO 20 *)
  END;  (* BuildSet *)

BEGIN
  (* Initialize both sets to null sets *)
  BySmaller := [];
  ByLarger := [];

  (* Build the two sets.
     BySmaller will contain 0, 3, 6, ..., 60;
     ByLarger will contain  0, 4, 8, ..., 80.
  *)
```

```
   BuildSet ( BySmaller, Smaller);
   BuildSet ( ByLarger, Larger);

   (* Display contents of the two sets *)
   ShowSetData ( BySmaller, 'BySmaller');
   ShowSetData ( ByLarger, 'ByLarger');

   (* Result is the intersection of BySmaller and ByLarger *)
   Result := BySmaller * ByLarger;
   (* Display contents of the intersection set *)
   ShowSetData ( Result, 'BySmaller * ByLarger');
   AwaitUser;
END.
```

Don't be intimidated by the program; it's really not complicated. The program uses the union operator to build two sets of values between 0 and 255, a subrange type. The sets are multiples of 3 and 4, respectively.

This work is done in procedure **BuildSet**, which provides an example of a **SET** parameter passed by reference. As when passing by value, you just specify the name of the set to be passed. In this case, however, the program will actually use the storage allocated for the argument. As a result, the changes will remain.

Procedure **ShowSetData** displays the contents of the specified set, along with a heading. After writing the heading, the procedure calls **DispByteSet**, which has the same structure as **DispCharSet**.

The intersection operator is used in the assignment to **Result**. Just put the operator (*) between two sets of the same type. If **BySmaller** consists of multiples of 3, and **ByLarger** consists of multiples of 4, then **Result** consists of multiples of 3 *and* 4. Mathematically, this is equivalent to multiples of 12, as you can see when **Result**'s elements are displayed. The program produces the following output:

```
BySmaller:
   3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
  48  51  54  57  60
ByLarger:
   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60
  64  68  72  76  80
BySmaller * ByLarger:
```

```
12  24  36  48  60
```

In this example, the intersection set has very few elements, compared to the operand sets. In the extreme, there may be no overlap between two sets. In such a case, the intersection set contains no elements. (Recall that such a set is known as the null set.) Two sets are called *disjoint* if their intersection is the null set.

In the following program, sets **SingleDigit** and **DoubleDigit** are disjoint:

Listing 7.20

```
(* Program to illustrate disjoint sets *)

PROGRAM DisjointDemo;

TYPE ByteRange = 0 .. 255;
     ByteSet = SET OF ByteRange;
VAR SingleDigit, DoubleDigit, Overlap : ByteSet;
    Count, NrIntersects : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  NrIntersects := 0;
  SingleDigit := [ 0 .. 9];
  DoubleDigit := [ 10 .. 99];
  Overlap := SingleDigit * DoubleDigit;
  WRITELN ( 'Overlap:');
  (* Display and count all common values.
     DispByteSet not used so NrIntersects could
     be used.
  *)
  FOR Count := 0 TO 255 DO
  BEGIN
    IF Count IN Overlap THEN
    BEGIN
      WRITE ( Count:4);
      INC ( NrIntersects);
    END; (* IF Count IN Overlap *)
  END; (* FOR Count ... *)
  WRITELN;
  WRITELN ( NrIntersects, ' common values');
  AwaitUser;
END.
```

The intersection set, **Overlap**, contains no elements. Therefore, no values will be displayed. The value of **NrIntersects** is displayed, to show that no common values were found.

As you've seen, the intersection of disjoint sets is the null set, by definition. What about the intersection of a null set with another set? You should be able to deduce this result: think about the size of an intersection set, compared with its operand sets.

TIP: The intersection of any set with the null set is always the null set itself. This is similar to multiplication by 0, which always yields 0.

The set intersection operator is a multiplicative operator, and has the same precedence as the multiplicative arithmetic and logical operators.

The Set Difference Operator

Suppose you want to check whether a student has taken all required courses. You could check each required course against the student's list of courses taken. If any course on the required list is not on the student's list, then the student is missing a required course.

Pascal's *set difference operator* (-) enables you to determine whether there are any required courses outstanding. This binary operator takes two sets (of the same type), and returns another set.

In the courses example, you would take the following set difference, in which **RequiredCourses** and **StudentsCourses** are both sets whose base types are course values:

```
RequiredCourses - StudentsCourses
```

If the result is anything but the null set, there are still required courses the student has not taken.

The following program shows how to use the set difference operator.

Listing 7.21

```
(* Program to illustrate set difference operator *)

PROGRAM DifferenceDemo;
USES Types;
CONST Smaller = 6;
      Larger = 8;
TYPE ByteRange = 0 .. 255;
     ByteSet = SET OF ByteRange;
VAR BySmaller, ByLarger, Result : ByteSet;
    Count : INTEGER;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Display any values between StartByte and EndByte,
     inclusive, that are elements in TheSet.
  *)
  PROCEDURE DispByteSet ( TheSet : ByteSet;
                        StartByte, EndByte : BYTE);
VAR IndexByte : BYTE;
    Count : INTEGER;
BEGIN
  Count := 1;
  FOR IndexByte := StartByte TO EndByte DO
    (* Note that IN takes a BYTE and a ByteSet
       as operands in this procedure
    *)
    IF IndexByte IN TheSet THEN
    BEGIN
      WRITE ( IndexByte:4);
      IF Count MOD 15 = 0 THEN
        WRITELN;
      INC ( Count);
    END;
END; (* DispByteSet *)

(* Display contents of the specified set. *)
PROCEDURE ShowSetData ( TheSet : ByteSet;
                        Message : STRING);
BEGIN
  WRITELN ( Message, ':');
  DispByteSet ( TheSet, 0, 255);
  WRITELN;
END;  (* Show SetData *)

(* Add the appropriate elements to TheSet:
   20 values, separated by Increment.
*)
PROCEDURE BuildSet ( VAR TheSet : ByteSet;
                     Increment : INTEGER);
VAR Count, Val : INTEGER;
BEGIN
  Val := 0;
  FOR Count := 1 TO 20 DO
  BEGIN
    Val := Val + Increment;
    TheSet := TheSet + [ Val];
  END; (* FOR Count := 0 TO 20 *)
  END;  (* BuildSet *)

BEGIN
  (* Initialize both sets to null sets *)
  BySmaller := [];
  ByLarger := [];

  (* Build the two sets.
     BySmaller will contain 0, 6, 12, ..., 120;
     ByLarger will contain  0, 8, 16, ..., 160.
  *)
  BuildSet ( BySmaller, Smaller);
  BuildSet ( ByLarger, Larger);

  (* Display contents of the two sets *)
  ShowSetData ( BySmaller, 'BySmaller');
  ShowSetData ( ByLarger, 'ByLarger');

  (* Result is the difference between BySmaller and ByLarger.
     That is, result contains those multiples of 6 that are
     NOT multiples of 8.
  *)
  Result := BySmaller - ByLarger;
  (* Display contents of the intersection set *)
  ShowSetData ( Result, 'BySmaller - ByLarger');
  AwaitUser;
END.
```

The program first builds two sets. One set contains multiples of 6, and the other contains multiple of 8. After displaying the contents of these two sets, the program removes the multiples of 8 from the set containing the multiples of 6. Thus, multiples of both 6 and 8 are removed from **BySmaller**.

This program is very similar to **IntersectionDemo**. The major difference is in the operator used when creating a value for **Result**, and the new values for **Smaller** and **Larger**.

The program produces the following output.

```
BySmaller:
   6  12  18  24  30  36  42  48  54  60  66  72  78  84  90
  96 102 108 114 120
ByLarger:
   8  16  24  32  40  48  56  64  72  80  88  96 104 112 120
 128 136 144 152 160
BySmaller - ByLarger:
   6  12  18  30  36  42  54  60  66  78  84  90 102 108 114
```

Notice that the difference set does not contain any multiples of 8. Essentially, the set difference operator removes from **BySmaller** any values that are also contained in **ByLarger**. The resulting set contains only elements unique to **BySmaller**.

If the two sets around a set difference operator are disjoint, then the set from which you're trying to remove elements is unchanged.

The following program shows how to remove a specified element from a set. The first part of the program removes two specific characters from the set. In the second part, you can remove whatever characters you wish.

Listing 7.22

```
(* Program to show how to remove single elements
   from a set.
*)

PROGRAM DiffDemo;

TYPE CharSet = SET OF CHAR;
VAR WdSet : CharSet;
    TheCh : CHAR;

   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Display any values between StartCh and EndCh, inclusive,
     that are elements in TheSet.
  *)
  PROCEDURE DispCharSet ( TheSet : CharSet;
                          StartCh, EndCh : CHAR);
  VAR IndexCh : CHAR;
      Count : INTEGER;
  BEGIN
    Count := 1;
    FOR IndexCh := StartCh TO EndCh DO
      (* Note that IN takes a CHAR and a CharSet
         as operands in this procedure
      *)
      IF IndexCh IN TheSet THEN
      BEGIN
        WRITE ( IndexCh:3);
        IF Count MOD 20 = 0 THEN
          WRITELN;
        INC ( Count);
      END;
    WRITELN;
  END; (* DispCharSet *)

  (* Prompt user and read a character *)
  PROCEDURE GetChar ( Message : STRING;
                      VAR Value : CHAR);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetChar *)

BEGIN  (* Main program *)
  WdSet := [ 'a', 'c' .. 'x', 'z'];
  DispCharSet ( WdSet, 'a', 'z');
  (* Remove 'a' from the set; this is done *)
  WdSet := WdSet - [ 'a'];
  (* Note how to write a single quote: include two quotes *)
  WRITELN ( 'After removing ''a''');
  DispCharSet ( WdSet, 'a', 'z');
  (* Remove 'b' from the set; nothing happens,
      since 'b' is not in the set.
  *)
  WdSet := WdSet - [ 'b'];
  WRITELN ( 'After removing ''b''');
  DispCharSet ( WdSet, 'a', 'z');
  GetChar ( 'Remove what char? (! to stop)', Thech);
  WHILE TheCh <> '!' DO
  BEGIN
    WdSet := WdSet - [ TheCh];
    WRITELN ( 'After removing ''', TheCh, '''');
    DispCharSet ( WdSet, 'a', 'z');
    GetChar ( 'Remove what char? (! to stop)', Thech);
  END; (* WHILE TheCh <> '!' *)
  AwaitUser;
END.
```

After building **WdSet**, the program removes two characters, *a* and *b*. This is accomplished by using the set constructor to create a set consisting of the single character, and then using the set difference operator to “subtract" this set's contents from **WdSet**.

Notice how to represent single quotes in the two “After removing" lines. To include a single quote within a string, you need to write *two* single quotes. thus, the quote before the letter is written as two quotes. The quote after the letter is also written as two quotes, and is then followed by a third quote which actually terminates the string argument.

This program produces the following output for the first part:

```
  a  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u
  v  w  x  z
After removing 'a'
  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v
  w  x  z
After removing 'b'
  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v
  w  x  z
```

Notice that there is no difference between the second and third versions of **WdSet**. Removing `b' from **WdSet** has no effect, since letter was not an element of the set to begin with.

The precedence for the set difference operator is the same as for the set union operator, since both are additive operators.

Comparing Sets

CodeWarrior Pascal has four operators for comparing sets. Comparisons between sets are based on the actual elements contained in each set, rather than on the number of elements in the sets.

For example, you can compare two sets to determine whether they have exactly the same collection of values as members. If so, the two sets are said to be *equal*.

You also can test whether one set (A) contains all the elements in the other set (B). If so, A is said to be a *superset* of B, and B is said to be a *subset* of A. (Note that A equals B is equivalent to saying that A is a superset of B and B is a superset of A.)

Like other comparisons, these end up as TRUE/FALSE questions. The result of such comparisons is — not surprisingly — a **BOOLEAN** value. The operands for such comparisons are sets of the same type.

TIP: Set comparison operators take two sets (of the same type) as arguments, and return a BOOLEAN value.

The Set Equality and Inequality Operators

Pascal's *set equality operator* (=) enables you to test whether its two operand sets contain exactly the same elements. If so, then the operator returns a **TRUE**; otherwise, the Boolean expression is **FALSE**. Note that the sets must contain the same elements, *not* just the same number of elements.

For example, suppose you have the following two sets:

```
Set1 := [ 'a', 'c' .. 'm', 'p']

Set2 := [ 'a' .. 'm', 'p' .. 's']
```

With these values, the following **IF** statement evaluates to **FALSE**:

```
IF Set1 = Set2 THEN
  ...
```

The *set inequality operator* (<>) enables you to test whether two sets are different.

The Subset Operator

In the previous example, all of **Set1**'s elements are also in **Set2**. This condition means that **Set1** is a *subset* of **Set2**.

In Pascal, you would use the *subset operator* for sets (<=) to indicate this relationship between the two sets. The following **IF** statement shows how you would test whether **Set1** is a subset of **Set2**:

```
IF Set1 <= Set2 THEN
  ...
```

The Boolean expression in this case evaluates to **TRUE**, for the example values.

The Superset Operator

Saying that **Set1** is a subset of **Set2** is equivalent to saying that **Set2** is a *super*set of **Set1**. In Pascal, the *superset operator* for sets (>=) enables you to test such an assertion. The following statement is equivalent to the one that used the subset operator.

```
IF Set2 >= Set1 THEN
  ...
```

TIP: Pascal has four comparison operators for sets: equality (=), inequality (<>), subset (<=), and superset (>=).

According to the definition of a subset, you'll see that two equal sets must be subsets of each other. For example, in the following program, all three **IF** clauses will be **TRUE** when **Set1** and **Set2** have the same elements.

Listing 7.23

```
(* Program to illustrate use of set comparison operators
   with equal sets.
*)

PROGRAM SetComparisonOpDemo;

TYPE CharSet = SET OF CHAR;

VAR Set1, Set2, Alphabet : CharSet;

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  Alphabet := [ 'a' .. 'z'];
  Set1 := [ 'a' .. 'l', 'o' .. 'z'];
  (* a very roundabout way of making
     Set2 equal to Set1.
     Alphabet - Set1 = [ 'm', 'n'], so
     Alphabet - [ 'm', 'n'] = Set1.
  *)
  Set2 := Alphabet - (Alphabet - Set1);
  IF Set1 = Set2 THEN
    WRITELN ( 'Set1 = Set2');
  IF Set1 <= Set2 THEN
    WRITELN ( 'Set1 is a subset of Set2');
  IF Set1 >= Set2 THEN
    WRITELN ( 'Set1 is a superset of Set2');
  AwaitUser;
END.
```

The program uses each of the set comparison operators to compare **Set1** and **Set2**. The statement that assigns a value to **Set2** is just an obscure way of making **Set1** and **Set2** contain the same element. If you're not sure how the assignment works, add **DispCharSet** to the program, and display the intermediate values.

There are no set comparison operators corresponding to strictly less than (<) and strictly greater than (>).

Table 7-1 summarizes the set operators:

Table 8: Pascal's Set Operators

Operator	Left Operand	Right Operand	Result
IN	value	set	BOOLEAN
Union (+)	set	set	set
Intersection (*)	set	set	set
Difference (-)	set	set	set
Equality (=)	set	set	BOOLEAN
Inequality (<>)	set	set	BOOLEAN
Subset (<=)	set	set	BOOLEAN
Superset (>=)	set	set	BOOLEAN

Summary

Summary

In this chapter, you learned about Pascal's three structured types: arrays, records, and sets. These types provide building blocks that you can combine and extend to suit the information in your program. The chapter also contained a brief preview of objects, which will be discussed in Chapter 11. In the next chapter, you'll learn about files.

8

Files

So far, the programs you've been building have involved the standard input (keyboard) and output (screen). Such programs get the information they need by reading what the user types in response to prompts. The programs then display their results on the screen. These displays disappear when new information is displayed or when another program starts running.

Sometimes its convenient to have a more permanent representation of the information that needs to be input or that will be displayed. In such a case, Pascal's files will come in handy. A *file* is essentially a way for a program to look at material from a disk in a logical manner — independent of the way the information may be laid out on the disk. For example, under page-oriented operating systems (such as DOS) a file's contents may be located at various physical sites on a disk.

As far as your program is concerned however, the file's contents are arranged in a neat, sequential manner. After the program reads an element from your file, it assumes that it can read the next element without any problem — as long as there are elements remaining in the file.

This is possible because of low-level resources (*drivers*), provided with the operating system and with your CodeWarrior Pascal environment. These resources take care of chores such as finding the next file element on the physical disk and making this next element available to your program's routines.

A file name will be associated with these contents. this name will be used when you need to refer to the file at the operating system command line or in a dialog box. For example, under DOS, you could give the following command to get directory information about a file named **DISKFILE.NAM**:

```
dir diskfile.nam
```

If such a file exists in your current directory, DOS will tell you its size, the date on which it was last modified, and so forth. Under Windows 95, the Explorer will give you similar kinds of information.

This chapter explains how to use files in your CodeWarrior Pascal programs. In particular, you'll learn how to create, open, and close files. You'll also learn how to read and write them. In the text, we'll use all uppercase characters to specify file names; in your listings you can use lowercase, uppercase, or mixed case.

Text Files

A file is actually a logical concept, which says nothing about the type of information contained in the file. In most cases, however, you'll want to store various kinds of textual material in your files. This might include strings or it might be numerical values. Such a *text file* contains ASCII information, and it generally offers no display surprises. You can read a text file with your text editor, and you can generally display it with a command such as **type** in DOS.

A text file is in contrast to a *binary file*, which is stored in a format that is unreadable for most text editors. As you'll see, the predefined procedures **WRITE**, **WRITELN**, **READ**, and **READLN** enable you to perform input and output with text files.

In Pascal, **TEXT** is a predefined type used to represent a text file. The following listing shows how you would declare a text file variable:

```
VAR FileVal : TEXT;
```

Actions on Files

When it comes to file handling, perhaps the two most important actions are opening and closing the files. Unless you do these two things properly, nothing else will work. Other actions on files include reading from them, writing to them, and moving to a specified location in a file.

Before you can use a file, you need to do two things: associate a file variable in your program with a file name and open the file. Different implementations of Pascal do this with either a single command or with a pair of commands. In its default setting, CodeWarrior Pascal does both of these things with a single command; other implementations use separate actions to link the variable with a name and then to open (or create) the file for use. For example, Borland's Turbo Pascal — arguably the most widely used

Pascal environment, and very popular in the DOS world — uses two statements.

You can use either method in CodeWarrior Pascal. To use two commands — that is, to make the I/O (input/output) compatible with Turbo Pascal, select the **Use Turbo Pascal I/O** option in the **Pascal Compiler** tab of the Project Settings dialog box.

The file variable is declared in the program, and does not exist beyond the program; the file name is known to the operating system and is used by it to identify the area(s) of storage that contain the file. Once this association between file variable and file name has been made, the operating system's drivers enable the interactions with the file. Predefined procedures **OPEN**, **RESET**, and **REWRITE** enable you to “hook" the file to the environment outside your program and also to make the file usable *within* the program.

CodeWarrior Pascal also includes predefined procedures to carry out actions on files once they've been opened or created. For example, there are predefined routines for reading from or writing to files, and also for moving to specific locations in files. Not all of these routines are available for all types of files, however. Last, but definitely not least, there is a predefined procedure for closing a file — thereby saving its contents on disk. In the following sections we'll look at files and the routines for using them in more detail.

Associating a Program and a Disk File

The **ASSIGN** procedure lets you associate a name with your file variable. For example, the following statement associates the name ZQZQ.QZQ with the **TEXT** variable **SampleFile**.

```
ASSIGN ( SampleFile, 'ZQZQ.QZQ');
```

The listing assumes you already have made the following variable declaration, and that the call to **ASSIGN** is within the scope of the declaration.

```
VAR SampleFile : Text;
```

The file name is specified as a string, whereas the file variable is represented by an identifier. The file variable and the file's name are used in two different contexts. Use the variable when you want to refer to the file within your program. For example, the following statement would write the number 395.7 to **SampleFile**:

```
WRITELN ( SampleFile, 395.7);
```

You'll learn more about such statements in the next section.

The file's name is not used within your program. Rather, the name is used to specify your file outside the program — for example on an operating system command line.

```
Type zqzq.qzq
```

This would be an instruction to display the contents of the file named ZQZQ.QZQ on the screen. DOS understands ZQZQ.QZQ, and interprets this as a reference to the file with that name stored on the disk.

On the other hand, the operating system would not understand the file variable identifier (**SampleFile**) used in the program, since the variable identifier disappears when your program stops executing. By associating the file name with the file variable through the **ASSIGN** call in your program, you provided a way of referring to your **Text** variable outside the program. Naturally, it's assumed that you actually saved the file to disk, by giving the appropriate command in your program.

Once you've "hooked" the file to the environment outside your program, you still need to make the file usable *within* the program. Procedures **RESET** and **REWRITE** enable you to accomplish this. (In standard Pascal, In the default settings for CodeWarrior Pascal, among other Pascal implementations, the calls to **ASSIGN** and to either **REWRITE** or **RESET** are combined — into a single call to either **REWRITE** or **RESET**.)

Creating a New File

REWRITE creates a new file; **RESET** opens a file that already exists on disk. You'll build a new file by writing to it, and you can read from an existing file. CodeWarrior Pascal also provides facilities for adding to an existing file.

Call up and run the following program, which shows how to create a new text file, and also how to write a single value into the file twice:

Listing 8.1

```
(* Program to illustrate use of REWRITE procedure; program will
   create a file named FileName, and will write a value into
   the file twice. In its current form, the file may be empty
   when the program finishes. Can you guess why?
*)

PROGRAM CreateDemo1;
```

```
CONST FileName = 'zwzw.wzw';
VAR SampleFile : Text;

BEGIN
   (* Associate the file variable (SampleFile) with a string,
      representing the file's name.
   *)
  ASSIGN ( SampleFile, FileName);
   (* Create a new file, which will be named FileName.
      Within the program, the file will be created using
      the next statement.
   *)
  REWRITE ( SampleFile);
  WRITELN ( SampleFile, 397.5);
  WRITELN ( SampleFile, 397.5 : 10 : 3);
  WRITELN ( 'Done');
  READLN;
END.
```

To write material in the file variable **SampleFile**, you can use the predefined **WRITE** and **WRITELN** procedures. When dealing with files, however, these procedures need a file variable as their first argument. The information to be written is specified in subsequent arguments. For example, the second call to **WRITELN** in the preceding program writes the value 397.5 to the file **SampleFile**.

If you compile this program using standard Pascal, and then do a directory listing after running the program, you'll find an entry for ZWZW.WZW. You may, however, find that this file is 0 bytes long and, if you look at such a file with your editor, you'll find that the file is empty. What happened to the 397.5s you wrote to the file?

TIP: When using WRITE or WRITELN to output information to text files, you must specify a file variable as the first argument.

In standard Pascal, the status of the file is undefined if it is not closed at the end of the program. Actually, most environments save the file contents along the way. In most cases, the worst that will happen is that the last item or items of information are not saved to the file.

Behind the scenes, a program works with a file buffer, which serves as a temporary storage area for material being written to a file. When the buffer is full, its contents are written to the file. A call to **CLOSE** empties the buffer one last time, and then instructs the operating system to close the file. In contrast, if the program ends without such a call, the buffer may not be emptied.

In order to save the contents of the entire file, you need to close the file explicitly. The predefined procedure **CLOSE** accomplishes this.

Listing 8.2

```
(* Program to illustrate use of REWRITE procedure; program will
   create a file named FileName, and will write a single value
   in the file.
*)

PROGRAM CreateDemo2;

CONST FileName = 'zwzw.wzw';
VAR SampleFile : Text;

BEGIN
  (* Associate the file variable (SampleFile) with a string,
     representing the file's name on disk.
  *)
  ASSIGN ( SampleFile, FileName);
  (* Create a new file, which will be named FileName.
     Within the program, the file will be created using
     the next statement.
  *)
  REWRITE ( SampleFile);
  WRITELN ( SampleFile, 397.5);
  WRITELN ( SampleFile, 397.5 : 10 : 3);
  (* close file, thereby saving contents *)
  CLOSE ( SampleFile);
  WRITELN ( 'Done');
  READLN;
END.
```

Now look at ZWZW.WZW with your editor. You'll find the number 397.5 (in exponential form and formatted) in the file. By calling **CLOSE**, you ensure that the buffer for the file is saved before the file is closed.

The variable declaration in the **CreateDemo2** program tells the compiler to allocate storage for a variable of type **Text**. This variable is stored internally as a record that contains various information about the file. The record also contains a buffer into which you can write information, and from which the information is transferred to the file. This buffer is emptied into the disk file. The size of the file can actually change during the course of the program.

Once declared, you can use **SampleFile** wherever a variable of type **Text** is allowed. This occurs in several places in the main program.

The predefined procedure **REWRITE** lets you create and open a new file. **REWRITE** takes two arguments: a file variable and a file name. When called, **REWRITE** opens the disk file associated with your file variable — with **SampleFile** in the program. This happens to be the file named ZWZW.WZW. In this particular case, the program actually creates a new file — regardless of whether a file named ZWZW.WZW already existed.

WARNING! REWRITE always creates a new file. If you already had a file with the specified name, this file would be lost, and would be replaced by a new, empty file.

Once the new file is opened, you can start writing to it. The predefined I/O routines (such as **WRITE** and **WRITELN**) are available. The same syntax holds as when you've used these routines in earlier programs. The only difference is that the first argument always needs to be a file variable. Subsequent arguments are used to specify what should be written — just as when you're writing to the standard output (the screen). The calls to **WRITELN** in the program illustrate how to write a value to a file.

After you've written what you want to the file, you need to call **CLOSE** to save your material. This call empties the buffer associated with the file, by writing the contents to the file. After doing this, the procedure closes the file, to save it permanently on disk.

The *file buffer* is an array of a particular size (128 bytes for a text file in CodeWarrior Pascal). Material intended for the file will be stored in this buffer. Once the buffer is full, the contents are "emptied" into the file. Explicit or implicit instructions to empty the buffer will accomplish the same thing. Either way, once it's emptied, the buffer is once again ready to receive information.

An Example

Let's look at one more example. The following program writes 125 random values to a file.

Listing 8.3

```
(* Program to illustrate file creation and use. Program:
     creates a file,  and writes 125 random values;
   The file's name will be the current value of FileName.
*)

PROGRAM CreateDemo3;
USES Windows;

CONST MaxTrials = 125;
      MaxPerLine = 5;
      FileName = 'zwzw.wzw';
VAR SampleFile : Text;
    Index, RandSeed : INTEGER;

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer value to seed the random number generator',
RandSeed);
  SRAND ( RandSeed);
  (* Associate the file variable (SampleFile) with a string,
     representing the file's name on disk.
  *)
  ASSIGN ( SampleFile, FileName);
  (* Create a new file, which will be named FileName. *)
  REWRITE ( SampleFile);
  FOR Index := 1 TO MaxTrials DO
  BEGIN
    (* write a value to the file *)
    WRITE ( SampleFile, RAND / MAXINT : 10 : 5);
    (* Every MaxPerLine values,
       move to a new line in the file
    *)
    IF Index MOD MaxPerLine = 0 THEN
      WRITELN ( SampleFile);
  END; (* FOR Index ... *)

  (* close file, thereby saving contents *)
  CLOSE ( SampleFile);
  WRITELN ( 'Done');
  READLN;
END.
```

Although the resulting file is much larger than for the previous program, **CreateDemo3** doesn't do anything new or different — when compared with **CreateDemo2**, for example.

The call to **WRITELN** might seem a bit unusual since it has only one argument: a file name. This is the file equivalent of simply calling **WRITELN** with no arguments (to send a CRLF) when not dealing with files.

The following listing shows the contents of ZWZW.WZW after a sample run:

```
   0.93334   0.96816   0.74464   0.61578   0.04921
   0.52520   0.30014   0.65408   0.59069   0.21203
   0.77887   0.00790   0.84718   0.36403   0.04982
   0.73287   0.40029   0.15667   0.07704   0.83261
   0.93488   0.09791   0.56901   0.44299   0.84783
   0.50981   0.57426   0.74785   0.51411   0.29629
   0.50770   0.04801   0.31945   0.54393   0.34193
   0.65551   0.81893   0.75256   0.94805   0.53492
   0.63102   0.74275   0.82015   0.86355   0.92624
```

```
0.53678   0.66809   0.93254   0.61998   0.76024
0.61267   0.06060   0.62882   0.36092   0.96787
0.44235   0.79961   0.69204   0.84428   0.10309
0.37089   0.25657   0.05818   0.38187   0.07582
0.65952   0.08076   0.61482   0.24613   0.56522
0.01165   0.94049   0.52645   0.70622   0.81129
0.39913   0.05826   0.24096   0.27647   0.74785
0.24309   0.00685   0.49338   0.66344   0.44405
0.46574   0.75258   0.25472   0.23534   0.38874
0.16027   0.05641   0.05889   0.57706   0.43026
0.92629   0.39765   0.16590   0.59088   0.51753
0.82485   0.30825   0.59140   0.24811   0.53858
0.28542   0.04220   0.93922   0.24117   0.34119
0.57310   0.54441   0.47260   0.78651   0.54425
0.64149   0.75740   0.17154   0.09390   0.60896
0.17443   0.57196   0.75748   0.65895   0.20885
```

Checking the Outcome of Your File Operation

The preceding programs will crash if, for some reason, a file cannot be created. For example, suppose you have no more space on your disk, so that a new file cannot be created.

Fortunately, CodeWarrior Pascal checks all calls to procedures such as **REWRITE**. If the procedure is unable to create the file, an error will be detected. By default, the program will terminate with a run-time error, and will display a message about the error.

This automatic checking and termination feature is nice because it protects you from certain kinds of errors that could cause problems. The automatic termination has disadvantages too, however. For example, it makes your programs less robust, since they will automatically crash if anything goes wrong in a call to a procedure involving files.

CodeWarrior Pascal has compiler directives for turning automatic checking off and on. Thus, you can check the outcome of procedure calls yourself, and can avoid the automatic termination in the case of an error.

The **{$I-}** directive turns automatic checking off. When this is off, the program does not crash if an I/O operation (such as creating a file) was unsuccessful. An error result is still returned, however.

You can check this value yourself by using the predefined function, **IOResult**. This function returns an **INTEGER**, which represents the outcome of the last I/O operation. If the operation was successful, the function returns a 0; otherwise some type of error occurred.

IOResult returns a meaningful result only if automatic checking is off. If an error has occurred while automatic checking is off, you must call **IOResult** before the program will respond to any more I/O operations.

To turn automatic checking (also known as *I/O checking*) back on, use **{$I+}**. The following program illustrates how to use these compiler directives to do your own checking on I/O operations. Note that this program will not crash — regardless of whether the file creation was successful.

Listing 8.4

```
(* Program to illustrate use of IOResult,
   and of compiler directives.
*)

PROGRAM IODemo;

CONST FileName = 'qqqq';
VAR Source : Text;

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN  (* Main Program *)
  ASSIGN ( Source, FileName);
  {$I-}                      (* Turn I/O checking off *)
  REWRITE ( Source);
  {$I+}                      (* Turn I/O checking back on *)
  IF IOResult = 0 THEN
    WRITELN ( 'Successful')
  ELSE
    WRITELN ( 'Could not create file ', FileName, '!');
  CLOSE (Source);
  AwaitUser;
END.
```

This program illustrates the general strategy for turning I/O checking on and off. Turn checking off before you call **REWRITE**, **RESET**, and other procedures available for dealing with files. After the call, you should generally turn automatic checking back on — using the **{$I+}** directive.

A Safe File Creation Procedure

The sequence of statements and directives in the preceding listing is very common if your programs work with files. If you do lots of file manipulation, a routine such as the one in the following listing will come in handy:

Listing 8.5

```
PROGRAM CheckFirstDemo;

VAR TheFile : Text;
    FName : STRING;

(* Create a new file, doing all necessary and warranted
   checking, and taking all precautions.
   Returns TRUE if file was created; FALSE otherwise.
*)
FUNCTION FMade ( VAR TheFile : Text;
                 FName : STRING) : BOOLEAN;
VAR FileExists : BOOLEAN;

BEGIN
  ASSIGN ( TheFile, FName);
  (* Check whether the file exists by trying to open it for
reading first *)
  {$I-}
  RESET ( TheFile);
  {$I+}
  IF IOResult = 0 THEN  (* If the file could be opened, it
exists. *)
  BEGIN
    FileExists := TRUE;
    WRITELN ( 'File already exists ... nothing to create.');
    (* IMPORTANT!!! Close the file, since we're not going to use
it. *)
    CLOSE (TheFile);
    FMade := FALSE;
  END  (* file exists *)
  ELSE   (* If the file doesn't exist, create it. *)
  BEGIN
    WRITELN ( 'File does not exist, will try to create it.');
    FileExists := FALSE;
    ASSIGN ( TheFile, FName);
    {$I-}
    REWRITE ( TheFile);
    {$I+}
    IF IOResult = 0 THEN
      FMade := TRUE
    ELSE
      FMade := FALSE;
  END;  (* file does not exist already *)
END; (* FN FMade *)

BEGIN (* Main *)
  WRITE ( 'Name of file to create? ');
  READLN (FName);
  WRITELN ( 'Calling FMade');
  IF FMade (TheFile, FName) THEN
  BEGIN
    WRITELN ( 'File ', FName, ' created ... writing a greeting
...');
    WRITELN ( TheFile, 'Greetings, curious reader. Welcome to ',
FName);
    CLOSE ( TheFile);
  END
  ELSE
```

```
    WRITELN ( 'No file created.');
  WRITE ( 'Press ENTER to end program.');
  READLN;
END.
```

By calling this function when you need to create a file, you simplify your source code: only one line is needed, instead of over half a dozen. You also make your program more robust and flexible. In case of an error, your program can take suitable actions.

Notice how to pass a text file as an argument to a routine. The format is exactly the same as for any other type of parameter — namely, <identifier> : <type specifier>

File parameters *must* be passed by reference, however. That is, the routine can never be expected to make a copy of the file. This is not feasible, because there is no way of knowing how large the file will be. The file could easily be too large to fit into available memory.

TIP: File parameters must be VAR parameters. You cannot pass a file by value.

Opening an Existing File

Call **RESET** instead of **REWRITE** if you already have a file with the specified name, and you want to keep the existing file, rather than replacing it with an empty one. The **RESET** procedure opens the file specified in its arguments. Unlike **REWRITE**, however, **RESET** assumes the specified file already exists, and is being opened for reading. In this example, we'll use the default CodeWarrior I/O — that is,
non-Turbo. Examine the instructions for opening the file, and compare these with the instructions you used in the earlier examples.

The following program reads the contents of file ZWZW.WZW, and performs some analyses of the values. Remember to turn off the **Use Turbo Pascal I/O** setting in the **Pascal Compiler** dialog box on
the Project Settings Menu.

Listing 8.6

```
(* Program to illustrate file opening and use. Program opens an
   existing file, and reads 125 random values contained in file.
   Each column of values is summed in a different array cell.
*)

PROGRAM ReadDemo1;

CONST MaxTrials = 125;
      MaxPerLine = 5;
      FileName = 'zwzw.wzw';
TYPE Stats = ARRAY [ 0 .. 4] OF REAL;
VAR SampleFile : Text;
    Index, WhichCell : INTEGER;
    Vals : Stats;  (* to store sums of columns *)
    Result : REAL;

  (* Open an existing file, doing all necessary
     and warranted checking, and taking all
     precautions.
     Returns TRUE if file was opened; FALSE otherwise.
     NOTE: Uses non-Turbo Pascal I/O
  *)
  FUNCTION cwFOpened ( VAR TheFile : Text;
                       FName : STRING) : BOOLEAN;
  BEGIN
    (* ASSIGN ( TheFile, FName);  *)
    (*$I-*)
    RESET ( TheFile, FName);
    (*$I+*)
    IF IOResult = 0 THEN
      cwFOpened := TRUE
    ELSE
      cwFOpened := FALSE;
  END; (* FN cwFOpened *)

  (* Initalize each cell of the Vals array to 0.0 *)
  PROCEDURE InitStats ( VAR Vals : Stats);
  VAR Index : INTEGER;
  BEGIN
    FOR Index := 0 TO 4 DO
      Vals [ Index] := 0.0;
  END; (* InitStats *)

  (* Display the results for each column separately. *)
  PROCEDURE DisplayMeans ( Vals : Stats;
                           ValsPerCell : INTEGER);
  VAR Index : INTEGER;
  BEGIN
    (* Display results for first four columns *)
    FOR Index := 1 TO 4 DO
      WRITELN ( Index, ': ',
                Vals [ Index] / ValsPerCell : 10 : 5);
    (* Display results for last column,
       the one corresponding to cell 0.
    *)
    WRITELN ( 5, ': ', Vals [ 0] / ValsPerCell : 10 : 5);
  END; (* DisplayMeans *)

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  InitStats ( Vals);
  IF cwFOpened ( SampleFile, FileName) = TRUE THEN
  BEGIN
    FOR Index := 1 TO MaxTrials DO
    BEGIN
      (* read a value from the file *)
      READ ( SampleFile, Result);
      (* Determine the column from which the current value
         was read.
      *)
      WhichCell := Index MOD MaxPerLine;
      (* Increment the appropriate cell by Result *)
      Vals [ WhichCell] := Vals [ WhichCell] +  Result;

      (* display current total in cell *)
      WRITE ( Vals [ WhichCell] : 10 : 5);
      IF Index MOD MaxPerLine = 0 THEN
        WRITELN;
    END; (* FOR Index ... *)

    DisplayMeans ( Vals, MaxTrials DIV MaxPerLine);

    CLOSE ( SampleFile);
  END (* IF File could be opened *)
  ELSE
    WRITELN ( 'Could not open file ', FileName);
  AwaitUser;
END.
```

The program reads all the values in ZWZW.WZW. These values must be read sequentially. That is, the second value in the file must be read after the first; the third must be after the second, and so forth.

This sequential method is the only way you can access information in text files. For this reason, text files are sometimes known as *sequential access structures*. This is in contrast to *random access structures* (such as arrays, or even typed files — which you'll see later in the chapter).

Each column of values is added separately. These sums are stored in a five-element array of **REAL**. While reading its information, the program displays the intermediate cell totals.

The program computes averages for each column. (These averages, or means, should be around 0.5, since the expression involving **SRAND** produces pseudorandom **REAL** values between 0.0 and 1.0).

In **ReadDemo1**, all file activity takes place in the main program and in **cwFOpened**. The two procedures are for initializing and displaying results on the screen.

Notice the call to **RESET** in function **cwFOpened**. This call ensures that the existing file, ZWZW.WZW, is used, instead of a new ZWZW.WZW being created for the run. Notice also that, in this function, all the work is done by the **RESET** procedure.

TIP: Files opened using RESET are input files, and are for reading only; files opened using REWRITE are output files, and are for writing only;

The call to **READ** in the main program shows how to get information from a text file. The predefined **READ** and **READLN** procedures are available. As with the output procedures, these routines work just as they do when you're reading form the standard input. The only difference is that the file variable again needs to be the first argument to **READ** and to **READLN**.

When **READ** is expecting an **INTEGER**, it skips over any intervening blanks, and then reads until it encounters a non-numerical character.

The procedure stops reading at that point. The next call to **READ** will continue reading from that position.

If the procedure encounters an end-of-line character (that is, if function **EOLN (SampleFile)** becomes **TRUE**), the procedure simply moves to the next line in the file. Subsequent values will be read beginning on the new line. If the procedure tries to read past the end of the file (that is, when **EOF(<FileName>)** is **TRUE**), the result is 0.

EOLN and **EOF** are predefined functions. When used in connection with files, these functions each take one argument — a file variable, and each returns a **BOOLEAN** that is **TRUE** if the end of the line (**EOLN**) or file (**EOF**) has been reached, and **FALSE** otherwise.

For the file generated by **CreateDemo3**, the program produces the following results:

```
0.93334   0.96816   0.74464   0.61578   0.04921
1.45854   1.26830   1.39872   1.20647   0.26124
2.23741   1.27620   2.24590   1.57050   0.31106
2.97028   1.67649   2.40257   1.64754   1.14367
3.90516   1.77440   2.97158   2.09053   1.99150
4.41497   2.34866   3.71943   2.60464   2.28779
4.92267   2.39667   4.03888   3.14857   2.62972
5.57818   3.21560   4.79144   4.09662   3.16464
6.20920   3.95835   5.61159   4.96017   4.09088
6.74598   4.62644   6.54413   5.58015   4.85112
7.35865   4.68704   7.17295   5.94107   5.81899
7.80100   5.48665   7.86499   6.78535   5.92208
8.17189   5.74322   7.92317   7.16722   5.99790
8.83141   5.82398   8.53799   7.41335   6.56312
8.84306   6.76447   9.06444   8.11957   7.37441
9.24219   6.82273   9.30540   8.39604   8.12226
9.48528   6.82958   9.79878   9.05948   8.56631
```

```
    9.95102    7.58216  10.05350    9.29482    8.95505
   10.11129    7.63857  10.11239    9.87188    9.38531
   11.03758    8.03622  10.27829   10.46276    9.90284
   11.86243    8.34447  10.86969   10.71087   10.44142
   12.14785    8.38667  11.80891   10.95204   10.78261
   12.72095    8.93108  12.28151   11.73855   11.32686
   13.36244    9.68848  12.45305   11.83245   11.93582
   13.53687   10.26044  13.21053   12.49140   12.14467
1:     0.54147
2:     0.41042
3:     0.52842
4:     0.49966
5:     0.48579
```

WARNING! If a file specified as a parameter does not exist, RESET causes a run-time error.

READ versus READLN when Dealing With Files

The following program is identical to **ReadDemo1** except that **ReadDemo2** uses **READLN** instead of **READ** in the main program's **FOR** loop. Another difference is that this version uses Turbo Pascal I/O.

Listing 8.7

```
(* Program to illustrate file opening and use.
   Program opens an existing file — named ZWZW.WZW on disk —
   and reads 125 random values contained in the file.
   Each column of values is summed in a different array cell.
   Program shows dangers of using READLN when you have
   multiple values on a line.
*)

PROGRAM ReadDemo2;
```

```
CONST MaxTrials = 125;
      MaxPerLine = 5;
      FileName = 'zwzw.wzw';
TYPE Stats = ARRAY [ 0 .. 4] OF REAL;
VAR SampleFile : Text;
    Index, WhichCell : INTEGER;
    Vals : Stats;  (* to store summs of columns *)
    Result : REAL;

   (* Open an existing file, doing all necessary
      and warranted checking, and taking all
      precautions.
      Returns TRUE if file was opened; FALSE otherwise.
  *)
  FUNCTION FOpened ( VAR TheFile : Text;
                     FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
   (*$I-*)
  RESET ( TheFile);
   (*$I+*)
  IF IOResult = 0 THEN
    FOpened := TRUE
  ELSE
    FOpened := FALSE;
  END; (* FN FOpened *)

   (* Initialize each cell of the array to 0.0 *)
  PROCEDURE InitStats ( VAR Vals : Stats);
  VAR Index : INTEGER;
  BEGIN
    FOR Index := 0 TO 4 DO
      Vals [ Index] := 0.0;
  END; (* InitStats *)

   (* Display the results for each column separately. *)
  PROCEDURE DisplayMeans ( Vals : Stats;
                           ValsPerCell : INTEGER);
  VAR Index : INTEGER;
  BEGIN
    (* Display results for first four columns *)
    FOR Index := 1 TO 4 DO
      WRITELN ( Index, ': ',
                Vals [ Index] / ValsPerCell : 10 : 5);
    (* Display results for last column,
       the one corresponding to cell 0.
    *)
    WRITELN ( 5, ': ', Vals [ 0] / ValsPerCell : 10 : 5);
  END; (* DisplayMeans *)

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN  (* Main program *)
  InitStats ( Vals);

  IF FOpened ( SampleFile, FileName) = TRUE THEN
  BEGIN
    FOR Index := 1 TO MaxTrials DO
    BEGIN
      (* read a value from the file, then move to next line *)
      READLN ( SampleFile, Result);
      WRITELN ( 'Result = ', Result : 10 : 5);
      (* Determine the column from which the current value
         was read.
      *)
      WhichCell := Index MOD MaxPerLine;
      (* Increment the appropriate cell by Result *)
      Vals [ WhichCell] := Vals [ WhichCell] +  Result;
      (* display current total in cell *)
      WRITE ( Vals [ WhichCell] : 10 : 5);
      IF Index MOD MaxPerLine = 0 THEN
        WRITELN;
    END; (* FOR Index ... *)
    DisplayMeans ( Vals, MaxTrials DIV MaxPerLine);

    CLOSE ( SampleFile);
  END; (* IF File exists *)
  AwaitUser;
END.
```

What do you think will happen in this program? Think about how **READLN** differs from **READ**.

Unlike **READ**, a call to **READLN** moves to the next line in the file after each value read. Thus, only one of the five values on each line is read. This means that, after 25 calls to **READLN**, the program will be at the end of the file (which contains only 25 lines). From then on, the program will be at the end of the file, and the calls to **READLN** will always return 0, which means the array cells won't change.

If you run this program, you'll get a listing such as the following, which shows that this is what happens.

```
0.93334   0.52520   0.77887   0.73287   0.93488
1.44315   1.03290   1.43438   1.36389   1.47166
2.05582   1.47525   1.80527   2.02341   1.48331
2.45495   1.71834   2.27101   2.18368   2.40960
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
3.27980   2.00376   2.84411   2.82517   2.58403
```

```
      3.27980   2.00376   2.84411   2.82517   2.58403
      3.27980   2.00376   2.84411   2.82517   2.58403
      3.27980   2.00376   2.84411   2.82517   2.58403
      3.27980   2.00376   2.84411   2.82517   2.58403
      3.27980   2.00376   2.84411   2.82517   2.58403
      3.27980   2.00376   2.84411   2.82517   2.58403
1:      0.13119
2:      0.08015
3:      0.11376
4:      0.11301
5:      0.10336
```

The fifth line of values is repeated for 20 more lines. Each of these last 100 values represent an attempt to read past the end of your file.

WARNING!   It may make a big difference whether you use READ or READLN to get information from a file. If you use READLN to read from a file with multiple values per line, you'll lose all but one of the values on each line.

Example: Removing TABs From a File

The following program copies each line of a file to another file. In the process of moving the line, each TAB character (ASCII 9) is replaced with an appropriate number of spaces.

The program also provides an opportunity to discuss the behavior of **READ** when single characters are being read.

Listing 8.8

```
(* Program to remove TAB characters from a file *)

PROGRAM DeTab;

CONST SmallCycle = 50;
```

```
      LargeCycle = 2500;
VAR Source, Log : Text;
    NrChRead : LONGINT;
    SName, LName : STRING;

  (* Create a new file, doing all necessary
     and warranted checking, and taking all
     precautions.
     Returns TRUE if file was created; FALSE otherwise.
  *)
  FUNCTION FMade ( VAR TheFile : Text;
                   FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
    (*$I-*)
    REWRITE ( TheFile);
    (*$I+*)
    IF IOResult = 0 THEN
      FMade := TRUE
    ELSE
      FMade := FALSE;
  END; (* FN FMade *)

  (* Open an existing file, doing all necessary
     and warranted checking, and taking all
     precautions.
     Returns TRUE if file was opened; FALSE otherwise.
  *)
  FUNCTION FOpened ( VAR TheFile : Text;
                     FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
  (*$I-*)
  RESET ( TheFile);
  (*$I+*)
  IF IOResult = 0 THEN
    FOpened := TRUE
  ELSE
    FOpened := FALSE;
  END; (* FN FOpened *)
```

```
(* Prompt user for a string *)
PROCEDURE GetString ( Message : STRING;
                      VAR Value : STRING);
BEGIN
  WRITE ( Message, ' ');
  READLN ( Value);
END; (* GetString *)

(* Write '.' periodically, to let the user know that
   the program is doing something.
   Val is the counter being tested;
   Small is the cycle for a '.'
   Large is the cycle for a WRITELN
*)
PROCEDURE ShowProgress ( Val, Small, Large : LONGINT);
CONST MarkerCh = '.';
BEGIN
  IF ( Val MOD Small = ( Small - 1)) THEN
    WRITE ( MarkerCh);
  IF ( Val MOD Large = ( Large - 1)) THEN
    WRITELN;
END; (* ShowProgress *)

(* Read InFile, and write it to OutFile.
   In the process, all TAB characters are replaced by an
   appropriate number of spaces.
   The procedure counts the number of characters processed.
*)
PROCEDURE DeTabFile ( VAR InFile, OutFile : Text;
                      VAR Count : LONGINT);
(* Give meaningful names to the characters that need special
   treatment.
   CtrlZ is the value returned when EOF is true.
*)
CONST TAB = #9;
      CR = #13;
      LF = #10;
      CtrlZ = #26;
      TabSkip = 8;
```

```
VAR Ch : CHAR;
    (* Index is used to test whether you're at a tabstop *)
    Index : INTEGER;

BEGIN  (* DeTabFile *)
  WRITELN ( 'In deTab');
  Index := 1;
  Count := 0;
  WHILE NOT EOF ( InFile) DO
  BEGIN
    (* Read a single character from InFile.
       This character can be any of the following:
       TAB, CR, LF, CtrlZ ( EOF), or anything else.
    *)
    READ ( InFile, Ch);
    CASE Ch OF
      TAB :     (* A TAB was read *)
        REPEAT
          WRITE ( OutFile, ' ');
          INC ( Index);
        UNTIL ( Index MOD TabSkip = 1);
      CR, LF :  (* either a CR or a LF was read *)
        BEGIN
          WRITE ( OutFile, Ch);
          Index := 1;
        END; (* CASE OF CR, LF *)
      CtrlZ : ; (* End of file has been reached; do nothing *)
      (* For anything else, just write the character to Log *)
      ELSE
        BEGIN
          WRITE ( OutFile, Ch);
          INC ( Index);
        END; (* CASE OF ELSE *)
    END; (* CASE OF Ch *)
    INC ( Count);
    ShowProgress ( Count, SmallCycle, LargeCycle);
  END; (* WHILE NOT EOF *)
  WRITELN ( 'LEAVING deTab');
END; (* DeTabFile *)
```

```
   (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN (* Main *)
  GetString ( 'Source File Name?', SName);
  GetString ( 'Log File Name?', LName);
  IF ( FOpened ( Source, SName) = TRUE) AND
     ( FMade ( Log, LName) = TRUE) THEN
  BEGIN
    WRITELN ( 'Getting Ready to detab file ', SName);
    DeTabFile ( Source, Log, NrChRead);
    WRITELN;
    WRITELN ( NrChRead, ' chars read');
    CLOSE ( Source);
    CLOSE ( Log);
  END; (* IF both files were opened *)
  AwaitUser;
END.
```

The program needs two file names. The first name refers to an existing file — presumably one that contains TAB characters. The contents of this file are read, character by character. Any required modifications are made to the material from the source file (**InFile**), and the modified material is written to **OutFile**, which is associated with the second file name you provide for the program.

The program's work is carried out by procedure **DeTabFile**. This routine reads each character from the file associated with **InFile**.

If the character is a TAB, the procedure replaces the TAB with the number of spaces needed to get to the next tabstop. (Tabstops are 8 spaces apart.) The value of **Index** is used to determine when a tabstop has been reached.

If the character is a carriage return (CR) or a line feed (LF), the characters is simply written to the target file. The tabstop counter is reset to 1, however, since the current file position is being changed to the leftmost column on a line.

Most other characters are simply passed through unchanged. One exception is the Ctrl-Z character (ASCII 26), which is returned when the end of the file has been reached, but which is also used by some text editors to pad out a file — for example, to make the file a particular size.

The **ShowProgress** procedure is used to show the user that the program is doing work. If a program has to do a task that takes a long time, there may be no output from the program for a while. This can be disconcerting to the user, since there is no way of knowing whether the program is working or has hung somewhere. In the program, **ShowProgress** writes a dot after every 50 characters read from **InFile**, and moves to a new line after every 1000 characters.

DeTabFile counts the number of characters read from the old file. This value is used as the first parameter for **ShowProgress**. Notice that the files are passed to **DeTabFile** just like any other argument would be. The procedure definition shows how to specify **Text** parameters. Note that these two parameters are **VAR** parameters, as required.

Reading Single Characters from a File

The **DeTab** program reads information from a file, character by character, using the predefined **READ** procedure. In the program, **READ** reads a single character from the specified file. The character read is returned in the second argument. For most values of this character, the procedure is done. A few characters (such as a carriage return) require special treatment, however.

The outcome when the character is read depends on whether either **EOLN** or **EOF** is **TRUE** — that is, whether you are at the end of a line in the file and whether you are at the end of the file itself.

If **EOF(InFile)** is true, then a Ctrl-Z is returned in the second parameter. If **EOLN(InFile)** is **TRUE**, then a carriage return (ASCII 13) is returned. In that case, the program will move to start of the next line in the file.

Notice that **EOLN** and **EOF** can take a file argument. By default, these functions apply to the standard input. However, just like many other predefined routines, **EOLN** and **EOF** can both take an additional parameter. If present, this argument must be a file variable.

Adding to An Existing File

As you know, **RESET** lets you open an existing file for reading, and that **REWRITE** lets you create a new file for writing. You can't write to a text file that was opened using **RESET**; similarly, you can't read from a text file that was created using **REWRITE**.

Sometimes, however, you may want to open an existing file, and add new material to the end of the file. The **APPEND** procedure enables you to do this. This procedure takes a single argument, of type **Text**. The procedure opens the file specified in the argument, and moves the current file position to the end of the file.

If the file does not exist, the procedure call results in a run-time error. If the file is already open, **APPEND** first closes the file and then reopens it. It's not good programming practice to call **APPEND** for a file that is already open.

The following program adds 125 more random values to ZWZW.WZW. The program calls **APPEND** to open the file for writing.

Listing 8.9

```
(* Program to illustrate use of APPEND procedure.
   Program opens a file — named ZWZW.WZW on disk —
   and adds 100 random values  (4 values per line) to
   the end of this file.
*)

PROGRAM AppendDemo1;
USES Windows;

CONST MaxTrials = 100;
      MaxPerLine = 4;
      FileName = 'zwzw.wzw';
VAR SampleFile : Text;
    Index, RandSeed : INTEGER;

  (* Open an existing file, doing all necessary
     and warranted checking, and taking all precautions.
     After the file is opened, the current file position is set
     to the end  of the file — so you can add more values.
     Returns TRUE if file was opened; FALSE otherwise.
  *)
  FUNCTION FAppended ( VAR TheFile : Text;
                           FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
    (*$I-*)
    APPEND ( TheFile);
    (*$I+*)
    IF IOResult = 0 THEN
      FAppended := TRUE
    ELSE
      FAppended := FALSE;
  END; (* FN FAppended *)

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer value to seed the random number generator',
RandSeed);
  SRAND ( RandSeed);
  IF FAppended ( SampleFile, FileName) = TRUE THEN
  BEGIN
    FOR Index := 1 TO MaxTrials DO
    BEGIN
      (* write a value to the file *)
      WRITE ( SampleFile, RAND / MAXINT : 10 : 5);
      (* Every MaxPerLine values,
         move to a new line in the file
      *)
      IF Index MOD MaxPerLine = 0 THEN
        WRITELN ( SampleFile);
    END; (* FOR Index ... *)

    CLOSE ( SampleFile);
  END
  ELSE  (* if call to APPEND was not successful *)
    WRITELN ( 'File ', FileName, ' could not be opened.');
  AwaitUser;
END.
```

Run the program, then look at file ZWZW.WZW. You'll find that this file now has 50 rows of values instead of 25. The call to **APPEND** (instead of **RESET**) enabled you to open an existing file and also to write to that file.

Operations on Text Files

Let's quickly summarize the things you can do with text files:

- You can use **ASSIGN** to associate a variable of type **Text** with a file on disk. You do this by specifying the disk file's name as the second argument to **ASSIGN**, with a **Text** variable as the first argument.
- You can use **REWRITE** to create a new file. If you already have a file with the same name as the new file, the existing file will be lost, and will be replaced by the new file. Such a file is for writing only.
- You can use **RESET** to open an existing file for reading. This file must exist, otherwise the program will crash with a run-time error — unless you have I/O checking turned off.
- You can use **APPEND** to open an existing file, for adding to the end of the file. A file opened with this procedure is for writing. When the file is opened, the current file position is the end of the file.
- You can use **CLOSE** to close a file on which you have been working. The call to this procedure adds any material remaining in the file's buffer to the actual file, and then closes the file to keep its contents on the disk.
- You can use **READ** and **READLN** to read material from the text file. This material can be of any simple type except enumerated types and **BOOLEAN**s. You can also read strings from the file.
- You can use **WRITE** and **WRITELN** to write material to the file. This material can be of any simple type except enumerated types You can also write **STRING**s to the

file.

- You can use function **IOResult** to check the outcome of an I/O operation. This function should be called only when I/O checking is off. Depending on the value returned by **IOResult** (0 if the operation was successful), you can take the appropriate actions in your program.
- You can use function **EOF** to determine whether your program has reached the end of the current file.
- You can use **EOLN** to determine whether you are at the end of the current line in a file.

That fairly modest list summarizes the kinds of things you can do with text files. Despite its small size, these capabilities enable you to do just about anything you'll need to do with a text file.

The major restriction on text files is that you must access the elements in your file sequentially. That is, you must read the first 10 lines of the file before you can get to the 11th line. The main reason for this is that the lines may not all be the same length. As a result, the program has no way of knowing where the 11th line starts — other than by reading the first 10 in order to skip them. In the next section, you'll learn how to define files that can be accessed randomly.

Typed Files

CodeWarrior Pascal also supports two kinds of non-text files: typed, or structured, and untyped. In a *typed file*, each element of the file is of the same type, and that type has a predefined, fixed size. The following program shows how to work with typed files. We'll look at it carefully in the following pages, to see how it works.

Listing 8.10

```
(* Program to illustrate use of a typed file -
   in this case, a File of INTEGER.
*)

PROGRAM TypedDemo;
USES Windows;
```

```
CONST FileName = 'qqqq.int';
      MaxTrials = 50;
  (* Describe the type of file you want to use *)
TYPE IFile = FILE OF INTEGER;

(* Declare a variable of the appropriate type *)
VAR SampleFile : IFile;
    RandSeed : INTEGER;

  (* Create a new IFile, doing all necessary and warranted
     checking, and taking all precautions.
     Returns TRUE if file was created; FALSE otherwise.
  *)
  FUNCTION iFMade ( VAR TheFile : IFile;
                    FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
    (*$I-*)
    REWRITE ( TheFile);
    (*$I+*)
    IF IOResult = 0 THEN
      iFMade := TRUE
    ELSE
      iFMade := FALSE;
  END; (* FN iFMade *)

  (* Open an existing IFile, doing all necessary and warranted
     checking, and taking all precautions.
     Returns TRUE if file was opened; FALSE otherwise.
  *)
  FUNCTION iFOpened ( VAR TheFile : IFile;
                       FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
  (*$I-*)
  RESET ( TheFile);
  (*$I+*)
  IF IOResult = 0 THEN
    iFOpened := TRUE
  ELSE
```

```
    iFOpened := FALSE;
  END; (* FN iFOpened *)

  (* Display integer values, in a specified # of columns,
     and displaying a specified # of values per line.
  *)
  PROCEDURE DispInts ( Value, Precision,
                       Count, NrPerLine : INTEGER);
  BEGIN
    WRITE ( Value : Precision);
    IF Count MOD NrPerLine = 0 THEN
      WRITELN;
  END;  (* DispInts *)

  (* Generate random integers, and store them in TheFile;
     call DispInts to show the values to the user as well.
  *)
  PROCEDURE GenerateInts ( VAR TheFile : IFile;
                           Nr : INTEGER);
  VAR Count, Result : INTEGER;
  BEGIN
    FOR Count := 1 TO Nr DO
    BEGIN
      Result := RAND MOD MAXINT;
      WRITE ( TheFile, Result);
      DispInts ( Result, 10, Count, 5);
    END; (* FOR Count ... *)
  END; (* GenerateInts *)

  (* Read all the values stored in the specified IFile,
     and show them to the user.
  *)
  PROCEDURE FReadInts ( VAR TheFile : IFile);
  VAR Count, Result : INTEGER;
  BEGIN
    Count := 0;
    WHILE NOT EOF ( TheFile) DO
    BEGIN
      READ ( TheFile, Result);
      INC ( Count);
      DispInts ( Result, 10, Count, 5);
    END; (* WHILE NOT EOF ... *)
  END; (* FReadInts *)

  (* Draw a line of a specified length,
     and using a specified symbol.
  *)
  PROCEDURE DrawLine ( Length : INTEGER;
                       Element : CHAR);
  VAR Count : INTEGER;
  BEGIN
    FOR Count := 1 TO Length DO
      WRITE ( Element);
    WRITELN;
  END; (* DrawLine *)

  (* Wait for user to press ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer value to seed the random number generator',
RandSeed);
  SRAND ( RandSeed);

  (* If the IFile was created, continue... *)
  IF iFMade ( SampleFile, FileName) THEN
  BEGIN
    GenerateInts ( SampleFile, MaxTrials);
    CLOSE ( SampleFile);  (* save the file's contents *)

    DrawLine ( 50, '-');

    (* If the file exists, read and display its contents. *)
    IF iFOpened ( SampleFile, FileName) THEN
    BEGIN
      FReadInts ( SampleFile);
      CLOSE ( SampleFile);
    END (* IF iFOpened *)
    ELSE
      WRITELN ( 'Could not open ', FileName);
  END
  ELSE
    WRITELN ( 'Could not create ', FileName);
  AwaitUser;
END.
```

Create the program and run it. Before you press ENTER, you'll see two copies of the same 50 values — separated by a dotted line.

The program generates and writes 50 pseudorandom **INTEGER**s to the specified file and to the screen. The program then reads each value from the file and displays the value. The file created by the program is 100 bytes long — 50 variables, each requiring 2 bytes.

After the program has finished running, try looking at QQQQ.INT with a text editor (for example, the CodeWarrior Pascal IDE editor). You'll see gibberish. This is because the file is not a text file. Rather, the contents are encoded in the format the program uses to store binary information. Since a typed file is not in ASCII format, you can't simply edit it.

There are advantages to typed files that might outweigh such a drawback. first, however, let's look at the program, to see how typed files are used. Don't be intimidated by the program's length. It is fairly straightforward, and you should have little difficulty following the discussion.

Defining a Typed File

If you want to use a **FILE OF INTEGER** in your program, you first have to describe this file type. That is, you need a type definition before you can declare variables of such a type.

The base type can be anything except a file type or a data structure that contains a file type as a component. For example, the following definitions are all valid. The comments indicate the size of each element in the file (in bytes). You'll find example programs that use some of these types later in the chapter.

```
(* 64 bytes : 31 bytes per string + 2 for INTEGER *)
TYPE Person = RECORD
                First, Last : STRING[ 30];
                Age : INTEGER;
              END; (* Person *)

     (* 40 bytes : 4 bytes per element *)
     LIStats = ARRAY [ 1 .. 10] OF LONGINT;

     (* 60 bytes : 6 bytes per element *)
     RStats = ARRAY [ 1 .. 10] OF REAL;

     (* 32 bytes : required to store 256 bits *)
     CharSet = SET OF CHAR;

(* Now define files containing elements of these types. *)
     PersFile = FILE OF Person;
     LIFile = FILE OF LIStats;
     RFile = FILE OF RStats;
     SetFile = FILE OF CharSet;
     IFile = FILE OF INTEGER;
```

Declaring a Typed File Variable

Once you've described the type of file you'll be using, you can declare variables of that type. In the program, the following line accomplishes this:

```
VAR SampleFile : IFile;
```

Creating and Opening Typed Files

Procedures **iFMade** and **iFOpened** serve, respectively, to create a new **IFile** and to open an existing one. The procedure definitions are almost identical to the definitions for **Fmade** and **FOpened**, which were defined for handling text files.

The only differences are in the first parameter. This indicates that you must handle typed files in the same manner as text files. That is, you first need to associate the file variable with an external file. After that, you need to open the file.

Similarly, the **CLOSE** procedure works the same way for typed files as it does for text files. To save the contents of a typed file, you need to close it.

Writing to Typed Files

Look at the main program for **TypedDemo**. Once the new **IFile** has been created in **TypedDemo**, the main program calls **GenerateInts**. This procedure generates the desired number of pseudorandom **INTEGER** values, and writes each of these values to the specified **IFile**. Notice that the information is written using the predefined **WRITE** procedure, as in the following statement from the program:

```
WRITE ( TheFile, TheResult);
```

This call has the same syntax as when writing to text files. In CodeWarrior Pascal, you can also use this procedure to write information to typed files. The system takes care of converting the information into the required form for the file.

While **WRITE** can be used with typed files, **WRITELN** cannot. This is because **WRITELN** adds a CRLF to the end of its output. This character (or character combination) is meaningless in a typed file. Including such characters in a **FILE OF INTEGER** would corrupt the file. As a result, **WRITELN** is not allowed for typed files.

WARNING! You cannot use WRITELN to output values to typed files; you can, however, use WRITE.

CodeWarrior Pascal's use of **WRITE** when outputting to typed files differs from standard Pascal. In standard Pascal, the process of writing information to a typed file is an involved process, and is beyond the scope of this book. However, you'll probably need to handle typed files differently if you switch from CodeWarrior Pascal to a different Pascal compiler.

Procedure **DispInts** checks that the information is formatted properly and displays the current value on the screen. The third parameter, **Count**, represents the number of values that have already been written. The procedure writes the **Value** passed as the first argument. Depending on whether **Count** is divisible by **NrPerLine**, the procedure may also call **WRITELN** to move to the next line on the screen.

Reading From Typed Files

After the specified number of values have been generated and written to the **IFile** by **GenerateInts**, SampleFile is closed. The file is reopened for reading; this is accomplished by calling **iFOpened**.

The actual reading is done in procedure **FReadInfo**. This procedure keeps reading values until it has read the last value in the file — at which point **EOF (TheFile)** becomes **TRUE** and the **WHILE** loop terminates. The **READ** procedure gets the value from the file.

The format for **READ** calls is the same whether you use text or typed files. When working with typed files, however, the second and later parameters must always be *variables* (not values) of the file's base type.

Each call to **READ** with **TheFile** as the first argument gets exactly one **INTEGER** value from **TheFile**. You *cannot* use **READLN** to get information from a typed file.

CodeWarrior Pascal also differs from standard Pascal in how values are read from a typed file. You cannot use **READ** to do this in standard Pascal. As was the case for outputting to typed files, the CodeWarrior Pascal approach is much simpler than the approach used in standard Pascal.

As each value is read from **TheFile**, the value is displayed on the screen using **DispInts**.

As you can see, the syntax for handling typed files is very similar to the syntax for dealing with text files. The contents of the files are in very different formats, however, as you should have noticed when you looked at QQQQ.INT.

Random Access to Typed Files

There is one very important difference between text and typed files. Text files are strictly sequential: you must read from the beginning of the file, and you must read each line in order.

When you have a typed file, in contrast, the program knows exactly how long each element is. Since there are no other characters (such as CRLF) in a typed file, the program should be able to move to any random element in the file — simply by computing an offset from the start of the file, and then moving the current file position to the location corresponding to this offset. In this context, an element is a value of the file's base type.

The predefined **SEEK** procedure enables you to move to an arbitrary element in a typed file. This procedure takes two arguments:

- The typed file in which to seek.
- The number corresponding to the element's position in the file. In CodeWarrior Pascal this value is a **LONGINT**.

In a typed file, elements are numbered beginning with 0. Thus, if a file had 1000 elements, these would be numbered 0 through 999. To move to the 10th element in a typed file — that is, the element with position 9 — you would use a call such as the following:

```
SEEK ( MyTypedFile, 9);
```

The following program shows how to use **SEEK**. The program writes a large number of pseudorandom **REAL**s to a file. After saving the file, the program reopens the file in order to read selected elements from the file.

The elements are specified by giving the position of the **REAL** in the file. All the elements displayed are from among the last 100 in the file — that is, elements with positions 1900 .. 1999. This restriction will make the advantages of random access very clear when compared against a file whose elements must be accessed sequentially.

Listing 8.11

```
(* Program to illustrate use of SEEK procedure. Program:
     writes MaxItems random values between 0.0 and 1.0
     to an RFile;
     reopens the file for reading; and
     reads 10 selected values from the file.
*)

PROGRAM SeekDemo;
USES Windows;

CONST MaxItems = 2000;
      FileName = 'qqqq.see';
TYPE RFile = FILE OF REAL;

VAR RF : RFile;
```

```
    Count, RandSeed, TestCount : INTEGER;
    WhichElem : LONGINT; (* SEEK takes a LONGINT parameter *)
    CurrVal : REAL;

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Create a new RFile, doing all necessary and warranted
     checking, and taking all precautions.
     Returns TRUE if file was created; FALSE otherwise.
  *)
  FUNCTION rFMade ( VAR TheFile : RFile;
                    FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
    (*$I-*)
(* !!!!!!!!!!!!!!!    REWRITE ( TheFile, FName); *)
    REWRITE ( TheFile);
    (*$I+*)
    IF IOResult = 0 THEN
      rFMade := TRUE
    ELSE
      rFMade := FALSE;
  END; (* FN rFMade *)

  (* Open an existing RFile, doing all necessary and warranted
     checking, and taking all precautions.
     Returns TRUE if file was opened; FALSE otherwise.
  *)
  FUNCTION rFOpened ( VAR TheFile : RFile;
                      FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
   (*$I-*)
  RESET ( TheFile);
(*  OPEN ( TheFile, FName); *)
   (*$I+*)
  IF IOResult = 0 THEN
    rFOpened := TRUE
  ELSE
    rFOpened := FALSE;
  END; (* FN rFOpened *)

BEGIN
  GetInt ( 'Integer value to seed the random number generator',
RandSeed);
  SRAND ( RandSeed);

  TestCount := 0;
  IF rFMade ( RF, FileName) THEN
  BEGIN
    WRITELN ( 'File ', FileName, ' successfully created. Writing
values ...');
     (* Put values into the file *)
    FOR Count := 1 TO MaxItems DO
    BEGIN
      CurrVal := RAND / MAXINT;
      WRITE ( RF, CurrVal);
      IF (Count < 10) OR (Count > 1900) THEN
      BEGIN
        WRITELN ( Count : 3, ': ', CurrVal : 10 : 5);
        READLN;
      END;
    END;
    CLOSE ( RF);
    WRITELN ( 'Done writing to file.');
     (* open the file again, to read from it *)
    IF rFOpened ( RF, FileName) THEN
    BEGIN
```

```
      WRITELN ( 'File ', FileName, ' successfully reopened.
...');
      FOR Count := 1 TO 10 DO
      BEGIN
        (* Select an element to get; this element will be among
           the last 100 in the file (1900 .. 1999);
        *)
        WhichElem := (RAND MOD 100) + 1900;
        (* Move current file position to the desired element *)
        SEEK ( RF, WhichElem);
        (* read the desired element *)
        READ ( RF, CurrVal);
        WRITELN ( WhichElem:5, ': ', CurrVal : 10 : 5);
      END;
      CLOSE ( RF);
    END (* IF rFOpened *)
    ELSE
      WRITELN ( 'Could not open ', FileName);
  END
  ELSE
    WRITELN ( 'Could not create ', FileName);
  AwaitUser;
END.
```

Run the program. You'll see that it takes much longer to create the file than to get the elements. Once the file is filled, it takes almost no time at all to move to and read a specified element in a typed file.

Notice how **SEEK** is used. After selecting an element, the value corresponding to this element's position in the file is passed as the second argument to **SEEK**. This call puts the current cursor position at the desired element. This means that the next value read from the file will be the desired element.

The seeking is always done from the beginning of the file, not from the current cursor position, so you don't need to close the file after each seek.

To compare this method to the way things work if you want to find the 1950*th* element (for example) in a text file, try running the following program. This program also writes a large number of values to a file, and then reads selected values from the file. This time, however, the file is of type **Text**. Such files are strictly sequential access files. In order to read the 1975*th* element, the program first needs to "read" over the preceding 1974 elements. This takes time, as you'll see when you run the program.

Listing 8.12

```
(* Program to illustrate how long it takes to
   move through a file sequentially.
*)

PROGRAM TextSeekDemo;
USES Windows;

CONST MaxItems = 2000;
      FileName = 'qqqq.xxx';

VAR RF : Text;
    Count, Index, RandSeed : INTEGER;
    WhichElem : INTEGER;
    CurrVal : REAL;

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Create a new text file, doing all necessary and warranted
      checking, and taking all precautions.
      Returns TRUE if file was created; FALSE otherwise.
  *)
  FUNCTION FMade ( VAR TheFile : Text;
                   FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
    (*$I-*)
    REWRITE ( TheFile);
    (*$I+*)
  IF IOResult = 0 THEN
    FMade := TRUE
  ELSE
    FMade := FALSE;
END; (* FN FMade *)

(* Open an existing text file, doing all necessary and
   warranted checking, and taking all precautions.
   Returns TRUE if file was opened; FALSE otherwise.
*)
FUNCTION FOpened ( VAR TheFile : Text;
                   FName : STRING) : BOOLEAN;
BEGIN
  ASSIGN ( TheFile, FName);
(*$I-*)
RESET ( TheFile);
(*$I+*)
IF IOResult = 0 THEN
  FOpened := TRUE
ELSE
  FOpened := FALSE;
END; (* FN FOpened *)

(* Display REAL values, in a specified # of columns,
   and displaying a specified # of values per line.
*)
PROCEDURE DispReals ( Value : REAL;
                      Width, Precision,
                      Count, NrPerLine : INTEGER);
BEGIN
  WRITE ( Value : Width : Precision);
  IF Count MOD NrPerLine = 0 THEN
    WRITELN;
END;  (* DispReals *)

PROCEDURE GetInt ( Message : STRING;
                   VAR Value : INTEGER);
BEGIN
  WRITE ( Message, '? ');
  READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer value to seed the random number generator',
RandSeed);
  SRAND ( RandSeed);

  IF FMade ( RF, FileName) THEN
  BEGIN
    (* Fill the file with random values. *)
    FOR Count := 1 TO MaxItems DO
    BEGIN
      CurrVal := RAND / MAXINT;
      WRITELN ( RF, CurrVal : 10 : 5);
    END;
    CLOSE ( RF);
    WRITELN ( 'Done');
    (* Read 10 specified elements from the file.
       All elements read are among the last 100 in the file.
       Note that the file must be closed and reopened each
       time. This is because calls to READ or READLN always
       move forward in the file.
    *)
    FOR Count := 1 TO 10 DO
    BEGIN
      IF FOpened ( RF, FileName) THEN
      BEGIN
        WhichElem := (RAND MOD 100) + 1900;
        (* Move through the file by reading each value *)
        FOR Index := 1 TO WhichElem DO
          READLN ( RF, CurrVal);
        WRITELN ( WhichElem : 5, ': ', CurrVal : 10 : 5);
        CLOSE ( RF);
      END  (* IF Fopened *)
      ELSE
        WRITELN ( 'Could not open ', FileName);
    END; (* FOR Count ... *)
  END
  ELSE
    WRITELN ( 'Could not create ', FileName);
  AwaitUser;
END.
```

Notice that the text file needs to be reopened for each value you want to read. This is because you can only go forward in a text file. Once you've read the 1950th element, you can't read the 1949th one without starting at the beginning of the file again.

To get to the beginning of a text file, you need to close the file and then use **RESET** to reopen it. The current file position will be at the start of the file when you first open it in this way.

The time required to close and reopen the file is negligible in comparison with the time required to move through the file. In short, it takes much longer to plow through 1949 values to get to the 1950*th* than it takes to jump immediately to the location of the 1950*th* element, and read the element.

More Typed File Examples

The programs in this section show some additional examples of typed files. They will give you some experience with a range of different types, making it easier for you to write your own programs. The example programs also show that you can write information (such as enumerated type values) to typed files that you cannot write to text files. These examples will also help you learn more about handling files.

Example: A File Containing Arrays

The following program shows how to define a file whose elements are 20-element arrays of **REAL**. This means that each “element" takes 80 (20 * 4) bytes.

Listing 8.13

```
(* Program to ilustrate use of files containing arrays. Program:
     writes MaxArrays arrays of random REAL values;
     reopens the typed file; and
     reads and displays the values in each array.
*)

PROGRAM ArrayTypedDemo;
```

```
USES Windows;

CONST MaxArrays = 3;
      ArrSize = 20;
      FileName = 'qqqq.rea';

(* Define an array type that will be the file's element *)
TYPE RStats = ARRAY [ 1 .. ArrSize] OF REAL;
      (* Define a typed file containing RStats elements *)
     RFile = FILE OF RStats;

VAR RF : RFile;
    RData : RStats;
    ArrCount, Count, RandSeed : INTEGER;
    NrElements : LONGINT;

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Create a new RFile, doing all necessary and warranted
     checking, and taking all precautions.
     Returns TRUE if file was created; FALSE otherwise.
  *)
  FUNCTION rFMade ( VAR TheFile : RFile;
                    FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
    (*$I-*)
    REWRITE ( TheFile);
    (*$I+*)
    IF IOResult = 0 THEN
      rFMade := TRUE
    ELSE
      rFMade := FALSE;
  END; (* FN rFMade *)
```

```
(* Open an existing RFile, doing all necessary and warranted
   checking, and taking all precautions.
   Returns TRUE if file was opened; FALSE otherwise.
*)
FUNCTION rFOpened ( VAR TheFile : RFile;
                    FName : STRING) : BOOLEAN;
BEGIN
  ASSIGN ( TheFile, FName);
(*$I-*)
RESET ( TheFile);
(*$I+*)
IF IOResult = 0 THEN
  rFOpened := TRUE
ELSE
  rFOpened := FALSE;
END; (* FN rFOpened *)

(* Display REAL values, in a specified # of columns,
   and displaying a specified # of values per line.
*)
PROCEDURE DispReals ( Value : REAL;
                      Width, Precision,
                      Count, NrPerLine : INTEGER);
BEGIN
  WRITE ( Value : Width : Precision);
  IF Count MOD NrPerLine = 0 THEN
    WRITELN;
END;  (* DispReals *)

(* Draw a line of a specified length,
   and using a specified symbol.
*)
PROCEDURE DrawLine ( Length : INTEGER;
                     Element : CHAR);
VAR Count : INTEGER;
BEGIN
  FOR Count := 1 TO Length DO
    WRITE ( Element);
  WRITELN;
END; (* DrawLine *)
```

```
  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer value to seed the random number generator',
RandSeed);
  SRAND ( RandSeed);

  (* Show the size of an RStats variable *)
  WRITELN ( 'RStats take ', SIZEOF ( RStats), ' bytes each');

  IF rFMade ( RF, FileName) THEN
  BEGIN
    WRITELN ( 'Writing arrays ...');
    FOR Count := 1 TO MaxArrays DO
    BEGIN
      (* Fill the array cells one by one ... *)
      FOR ArrCount := 1 TO ArrSize DO
        RData [ ArrCount] := RAND / MAXINT;
      (* ... but write the entire array to the file at once *)
      WRITE ( RF, RData);
    END;
    CLOSE ( RF);
    WRITELN ( 'Done writing.');

    (* reopen the file for reading *)
    IF rFOpened ( RF, FileName) THEN
    BEGIN
      WHILE NOT EOF ( RF) DO
      BEGIN
        DrawLine ( 50, '-');
        (* read an entire array at a time *)
        READ ( RF, RData);
        INC ( ArrCount);
        (* dislpay the array elements, one by one *)
        FOR ArrCount := 1 TO ArrSize DO
          DispReals ( RData [ ArrCount], 10, 5, ArrCount, 5);
      END;
      DrawLine ( 50, '-');
    END  (* IF rFOpened *)
    ELSE
      WRITELN ( 'Could not open ', FileName);
    (* Display information about the # of elements in file *)
    NrElements := FileSize ( RF);
    WRITELN ( FileName, ' contains ', NrElements,
              ' elements (arrays).');
    CLOSE ( RF);
  END  (* IF rFMade *)
  ELSE
    WRITELN ( 'Could not create ', FileName);
  AwaitUser;
END.
```

After the definition for **RStats**, the file description is completely straightforward. The **rFMade** and **rFOpened** procedures are similar to routines you've seen in earlier programs, except for the first parameters.

You've already seen the other routines — **DrawLine**, DispReals, and **AwaitUser**. They should not interfere with your overview of the program.

Note how the file elements and the file are used in the main program. Keep in mind that each element actually contains 20 smaller components. Therefore, before you can write an element to the **RFile**, you need to initialize the array — that is, the element itself.

This means initializing the 20 cells, which requires a **FOR** loop. Once the 20 iterations have been completed, the entire array (all 80 bytes) is written to the typed file. This process is repeated: the second array is initialized, and is then written to the file.

Notice that the calls to **WRITE** and **READ** are no different than for simpler base types. The major difference is simply that multiple items of information can be transferred at once — because they are all part of a single, structured type, which happens to be the file's base type. If the file were of type **Text**, you would not be able to do this.

Similarly, an entire **RFile** array is read at once from the typed file. Once the array has been read into the program, the individual cells of the array must be displayed one by one.

Finally, the last part of the outer **IF** clause introduces a new predefined function: **FileSize**. This function takes a typed file argument, and returns the number of *elements* (not bytes) in the file. In the example program, this value is 3, since the file only contains three (**RStats**) arrays — despite the fact that it actually contains 60 values.

Example: Record and Set Types as File Components

You can have typed files with more exotic types as their components. The following program uses a record as the basic component. One field of this record is a **CharSet** — a **SET OF CHAR**.

Run the program, to see storage information about **CharSet** and **ChRec**, and to determine the number of elements in the typed file. By running the program, you'll also see the random **CharSet**s generated by the program.

Listing 8.14

```
(* Program to ilustrate use of records containing sets
   as a component type for a typed file. The program:
     displays information about the size of a ChRec;
     creates a file to contain ChRec values;
     determines maximum # of elements for current set;
     generates MaxTrials sets of random CHAR values;
     writes each of these sets to the typed file;
     reopens the typed file for reading;
     reads the contents of the file one element
       (that is, ChRec) at a time;
     displays the contents of each ChRec;
*)

PROGRAM ChRecTypedDemo;
USES Windows;

CONST MaxTrials = 10;
      FileName = 'qqqq.crc';
TYPE CharSet = SET OF CHAR;

     ChRec = RECORD
               SetInfo : CharSet;
               MaxItems : INTEGER;
```

```
            END;

      ChRecFile = FILE OF ChRec;

VAR ChRF : ChRecFile;
    ChRecData : ChRec;
    SetCount, NrTimes, ArrCount, Count, RandSeed : INTEGER;
    ChIndex : CHAR;

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Draw a line of a specified length,
     and using a specified symbol.
  *)
  PROCEDURE DrawLine ( Length : INTEGER;
                       Element : CHAR);
  VAR Count : INTEGER;
  BEGIN
    FOR Count := 1 TO Length DO
      WRITE ( Element);
    WRITELN;
  END; (* DrawLine *)

  (* Create a new ChRecFile, doing all necessary and warranted
     checking, and taking all precautions.
     Returns TRUE if file was created; FALSE otherwise.
  *)
  FUNCTION chrFMade ( VAR TheFile : ChRecFile;
                      FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
    (*$I-*)
    REWRITE ( TheFile);
    (*$I+*)
    IF IOResult = 0 THEN
```

```
      chrFMade := TRUE
    ELSE
      chrFMade := FALSE;
  END; (* FN chrFMade *)

  (* Open an existing ChRecFile, doing all necessary and
     warranted checking, and taking all precautions.
     Returns TRUE if file was opened; FALSE otherwise.
  *)
  FUNCTION chrFOpened ( VAR TheFile : ChRecFile;
                        FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
  (*$I-*)
  RESET ( TheFile);
  (*$I+*)
  IF IOResult = 0 THEN
    chrFOpened := TRUE
  ELSE
    chrFOpened := FALSE;
  END; (* FN chrFOpened *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer value to seed the random number generator',
RandSeed);
  SRAND ( RandSeed);

  (* display information about storage requirements *)
  WRITELN ( 'CharSets take ', SIZEOF ( CharSet), ' bytes');
  WRITELN ( 'ChRecs take ', SIZEOF ( ChRec), ' bytes');

  IF chrFMade ( ChRF, FileName) THEN
  BEGIN
```

```
(* build 10 sets, and write each to ChRF *)
FOR Count := 1 TO MaxTrials DO
BEGIN
  (* Determine the # of candidate elements for the set *)
  NrTimes := RAND MOD 27;
  ChRecData.SetInfo := [];   (* initialize to empty set *)
  ChRecData.MaxItems := NrTimes;
  (* generate NrTimes random letters; try to add to set *)
  FOR SetCount := 1 TO NrTimes DO
    (* the right hand side ensures that only
       'a' .. 'z' are used.
    *)
    WITH ChRecData DO
      SetInfo := SetInfo + [ CHR ( (RAND MOD 26) + 97)];
  (* after building the set component element-by-element,
     write the entire record at once.
  *)
  WRITE ( ChRF, ChRecData);
END;
CLOSE ( ChRF);

DrawLine ( 50, '-');

IF chrFOpened ( ChRF, FileName) THEN
BEGIN
  WHILE NOT EOF ( ChRF) DO
  BEGIN
    (* Read an entire ChRec at a time *)
    READ ( ChRF, ChRecData);
    WRITE ( 'Max = ', ChRecData.MaxItems, ' : ');
    (* Write each set element *)
    FOR ChIndex := 'a' TO 'z' DO
    BEGIN
      IF ChIndex IN ChRecData.SetInfo THEN
        WRITE ( ChIndex, ' ');
    END;
    WRITELN;
  END;  (* WHILE NOT EOF *)
  DrawLine ( 50, '-');
  WRITELN ( FileName, ' contains ', FileSize ( ChRF),
                ' elements (ChRecs)');

    END   (* IF chrFOpened *)
    ELSE
      WRITELN ( 'Could not open ', FileName);
  END   (* IF chrFMade *)
  ELSE
    WRITELN ( 'Could not create ', FileName);
  AwaitUser;
END.
```

Once you've defined all your types, and have declared the appropriate variables, the syntax for the file handling is identical in all the examples. This means that you only need to change the details of routines such as **FMade** and **FOpened** to create the appropriate versions.

You may also need to change the details of how the file components get their values. For example, assigning values to a **ChRec** is much more involved than assigning values to an **INTEGER**. Nevertheless, once the variables have their values, you'll use the same simple procedure calls to write the information to a typed file.

Example: Files Containing Enumerated Types

You can even define files whose base type is an enumerated type. Recall that you could not write the values of enumerated types directly. This is true if you're trying to write to the screen or to a text file.

On the other hand, if you want to write to a typed file, it is possible to store values for enumerated types. This is because the ordinal values are stored. If you want to display these values on the screen, however, you'll have to make the appropriate substitutions.

The following program shows an example involving typed files. Since the file handling aspects of the program are the same as for earlier programs, the program is presented without comment. You may, however, want to experiment with the program. For example, try appending more values to the file. You'll need to define a procedure for calling **APPEND** (which works for typed files) in the appropriate manner.

Listing 8.15

```
(* Program to illustrate files with enumerated types
   as components.
*)

PROGRAM EnTypeDemo;

CONST FileName = 'qqqq.clr';
TYPE Color = ( red, green, blue, yellow, white, black);
     CFile = FILE OF Color;
VAR CF : CFile;
    CurrColor : Color;

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Draw a line of a specified length,
     and using a specified symbol.
  *)
  PROCEDURE DrawLine ( Length : INTEGER;
                       Element : CHAR);
  VAR Count : INTEGER;
  BEGIN
    FOR Count := 1 TO Length DO
      WRITE ( Element);
    WRITELN;
  END; (* DrawLine *)

  (* Create a new CFile, doing all necessary and warranted
     checking, and taking all precautions.
     Returns TRUE if file was created; FALSE otherwise.
  *)
  FUNCTION cFMade ( VAR TheFile : CFile;
                    FName : STRING) : BOOLEAN;
  BEGIN
```

```
    ASSIGN ( TheFile, FName);
    (*$I-*)
    REWRITE ( TheFile);
    (*$I+*)
    IF IOResult = 0 THEN
      cFMade := TRUE
    ELSE
      cFMade := FALSE;
  END; (* FN cFMade *)

  (* Open an existing CFile, doing all necessary and
     warranted checking, and taking all precautions.
     Returns TRUE if file was opened; FALSE otherwise.
  *)
  FUNCTION cFOpened ( VAR TheFile : CFile;
                      FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
  (*$I-*)
  RESET ( TheFile);
  (*$I+*)
  IF IOResult = 0 THEN
    cFOpened := TRUE
  ELSE
    cFOpened := FALSE;
  END; (* FN cFOpened *)

BEGIN  (* Main Program *)
  IF cFMade ( CF, FileName) THEN
  BEGIN
    WRITELN ( 'File ', FileName, ' created successfully. Writing
...');
    (* CurrColor actually is assigned an ordinal value *)
    CurrColor := red;
    (* Write this ordinal value to CF *)
    WRITE ( CF, CurrColor);
    CurrColor := black;
    WRITE ( CF, CurrColor);
    CurrColor := green;
    WRITE ( CF, CurrColor);
    CurrColor := white;
    WRITE ( CF, CurrColor);
    CurrColor := blue;
    WRITE ( CF, CurrColor);
    CurrColor := yellow;
    WRITE ( CF, CurrColor);
    CLOSE ( CF);

    IF cFOpened ( CF, FileName) THEN
    BEGIN
      WRITELN ( 'File ', FileName, ' opened successfully.');
      WHILE NOT EOF ( CF) DO
      BEGIN
        (* Read an ordinal value from CF *)
        READ ( CF, CurrColor);
        (* Display the string appropriate to the
           ordinal value.
        *)
        CASE CurrColor OF
          red : WRITELN ( 'red');
          green : WRITELN ( 'green');
          blue : WRITELN ( 'blue');
          yellow : WRITELN ( 'yellow');
          white : WRITELN ( 'white');
          black : WRITELN ( 'black');
        END; (* CASE of CurrColor *)
      END; (* WHILE NOT EOF *)
      CLOSE ( CF);
    END   (* IF cFOpened *)
    ELSE
      WRITELN ( 'Could not open ' , FileName);
  END  (* IF cFMade *)
  ELSE
    WRITELN ( 'Could not create ' , FileName);
  AwaitUser;
END.
```

Untyped Files

Recall that CodeWarrior Pascal also supports untyped files. To declare an untyped file, you simply specify **FILE** as the variable's type. An untyped file is essentially a stream of bytes. Your program transfers chunks of information between program and file. The chunks are of a fixed size, which you specify when you create or open the file.

To write information into an untyped file, your program simply transfers the appropriate number of bytes to the file. Similarly, when reading from the file, the program transfers bytes from the file. The bytes are interpreted in the program.

WRITE and **READ** will not work for untyped files, however. Instead, you need to use two other routines: **BlockRead** and **BlockWrite**. These are discussed briefly after the following program, which illustrates the use of untyped files. The program generates a few random **REAL**s and writes their bit patterns to an untyped file. The information is transferred 4 bytes at a time, since that's the size of a **REAL**.

After the file has been written and saved, the program reads information back from the file, and displays the values on the screen. Run the program to see how it works.

Listing 8.16

```
(* Program to ilustrate use of untyped files.
   Program generates random REALs, and writes these
   to an untyped file. The information is transferred
   4 bytes at a time, since that's the size of a REAL.
   NOTE that input and output is done with BlockRead and
   BlockWrite.
*)

PROGRAM UntypedDemo;
USES Windows;

CONST MaxTrials = 10;
      FileName = 'qqqq.unt';
VAR UT : FILE;
    Count, RandSeed : INTEGER;
    Val : REAL;
```

```
PROCEDURE AwaitUser;
VAR Ch : CHAR;
BEGIN
  WRITE ( 'Press ENTER to continue.');
  READLN ( Ch);
END; (* AwaitUser *)

(* Draw a line of a specified length,
   and using a specified symbol.
*)
PROCEDURE DrawLine ( Length : INTEGER;
                     Element : CHAR);
VAR Count : INTEGER;
BEGIN
  FOR Count := 1 TO Length DO
    WRITE ( Element);
  WRITELN;
END; (* DrawLine *)

(* Create a new ChRecFile, doing all necessary and warranted
   checking, and taking all precautions.
   Returns TRUE if file was created; FALSE otherwise.
*)
FUNCTION utFMade ( VAR TheFile : FILE;
                   FName : STRING;
                   ChunkSize : WORD) : BOOLEAN;
BEGIN
  ASSIGN ( TheFile, FName);
  (*$I-*)
  REWRITE ( TheFile, ChunkSize);
  (*$I+*)
  IF IOResult = 0 THEN
    utFMade := TRUE
  ELSE
    utFMade := FALSE;
END; (* FN utFMade *)

(* Open an existing ChRecFile, doing all necessary and
   warranted checking, and taking all precautions.
   Returns TRUE if file was opened; FALSE otherwise.
  *)
  FUNCTION utFOpened ( VAR TheFile : FILE;
                       FName : STRING;
                       ChunkSize : WORD) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
  (*$I-*)
  RESET ( TheFile, ChunkSize);
  (*$I+*)
  IF IOResult = 0 THEN
    utFOpened := TRUE
  ELSE
    utFOpened := FALSE;
  END; (* FN utFOpened *)

  (* Display REAL values, in a specified # of columns,
     and displaying a specified # of values per line.
  *)
  PROCEDURE DispReals ( Value : REAL;
                        Width, Precision,
                        Count, NrPerLine : INTEGER);
  BEGIN
    WRITE ( Value : Width : Precision);
    IF Count MOD NrPerLine = 0 THEN
      WRITELN;
  END;  (* DispReals *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer value to seed the random number generator',
RandSeed);
  SRAND ( RandSeed);

  (* Will write REALs *)
  IF utFMade ( UT, FileName, SIZEOF (REAL)) THEN
  BEGIN
    (* generate random values, and write each to utF *)
    FOR Count := 1 TO MaxTrials DO
    BEGIN
      Val := RAND / MAXINT;
      (* This writes one REAL-sized chunk to UT.
         Parameters: File, Buffer (Variable),
                     # of chunks being transferred.
      *)
      BlockWrite ( UT, Val, 1);
      DispReals ( Val, 10, 5, Count, 5);
    END;
    CLOSE ( UT);

    DrawLine ( 50, '-');

    IF utFOpened ( UT, FileName, SIZEOF (REAL) ) THEN
    BEGIN
      WHILE NOT EOF ( UT) DO
      BEGIN
        (* Read one REAL-sized chunk from the file *)
        BlockRead ( UT, Val, 1);
        INC ( Count);
        DispReals ( Val, 10, 5, Count, 5);
      END;  (* WHILE NOT EOF *)
      DrawLine ( 50, '-');
      AwaitUser;
      CLOSE ( UT);
    END   (* IF chrFOpened *)
    ELSE
      WRITELN ( 'Could not open ', FileName);
  END   (* IF chrFMade *)
  ELSE
    WRITELN ( 'Could not create ', FileName);
END.
```

Before studying the main program, look at **utFMade** and **utFOpened**. These routines have one parameter more than their counterparts for typed files. This third parameter specifies the number of bytes to be treated as a single record, or chunk, when transferring information to and from the file.

For example, in the program, this argument is 4 when these procedures are called. This value allows you to transfer an entire **REAL** at a time, between program and file. The default chunk size is 128 bytes. However, you should select your chunk size on the basis of the type of information you actually do have in the file.

Thus, if you have **INTEGER** values, it makes sense to transfer two bytes at a time. On the other hand, if your file contains a mixture of types, your safest strategy is to transfer the smallest unit — one byte at a time. Your program would then need to combine bytes in order to build the types of values it needed.

The **ChunkSize** parameter is used in the calls to **RESET** and **REWRITE** in **utFOpened** and **utFMade**, respectively. The calls to **RESET** and **REWRITE** have one parameter more then they did for text and typed files.

To write information to an untyped file, you need to use the **BlockWrite** procedure. This routine takes three arguments and may take a fourth if you wish.

- The first argument is an untyped file variable.
- The second argument can be any variable, and represents a memory location. The procedure interprets this as the starting location of a buffer whose size is determined by the third parameter. This parameter is passed by reference.
- The third argument is an **INTEGER**, and specifies the number of chunks to read at a time. For example, if this value were 2 in the program, then each call to **BlockWrite** would transfer 8 bytes to the untyped file. This parameter is passed by value.
- You may include a fourth parameter (also an **INTEGER**) which provides information about the number of chunks actually written. After **BlockWrite** has finished its work, the value of this parameter will represent the number of chunks written. This number will be less than or equal to the number of chunks you specified as the third argument to **BlockWrite**.

For example, the disk may become full before **BlockWrite** has transferred all the elements you specified in the third parameter. If so, fewer chunks than you specified will have been transferred. To determine how many were transferred, you have to check the value of the fourth parameter. (Naturally, you need to specify a variable as your fourth parameter when calling the procedure.)

When **BlockWrite** does its work, the current file position is updated accordingly. Essentially, current file position will always be at the end of the file, since new material is always added to the end of a file.

Procedure **BlockRead** has the same parameter structure as **BlockWrite**. This procedure reads the specified number of chunks from the untyped file, and stores these bytes at the memory location specified by the second parameter, which can be any variable.

For these procedures to work, the untyped file must be open.

Miscellaneous File Handling Routines

There are other procedures and functions that can take file arguments. For example, there are procedures for renaming or deleting files, for checking the current directory on a specified drive, for checking the current position in a non-text file, and so forth.

The situations in which you might use such routines are more advanced than we can look at here. In fact, some of the actions (such as erasing a file) can get you into real trouble if you're not careful. If you're interested in such topics, you'll need to consult the documentation for the Pascal library included with the CodeWarrior Pascal programming environment.

Summary

In this chapter you learned about files in Pascal, and how to work with them. Most commonly, you'll use text files, which were discussed in the first half of the chapter. Sometimes, however, you may find it useful to work with typed or even untyped files. Non-text files are random access, which makes them useful if you have lots of data that you might need to access.

In the next chapter, you'll learn about Pascal's pointer types.

9

Pointers

In this chapter you'll learn about Pascal's pointer data type. This type is generally used to access storage that is allocated when your program is executing, rather than in advance. Being able to allocate storage at run-time, rather than at compile-time, enables you to do things that would otherwise be difficult or impossible to do efficiently. For example, by allocating storage at run-time, you can handle tasks for which you don't know in advance how many values you'll have to process.

Static Memory Allocation

When you declare variables in a Pascal program, the compiler knows exactly how much storage is required for them. The compiler allocates storage for global variables and constants in the form of a *data segment* --- an area of contiguous memory whose size is determined by the number and types of global variables declared in the program.

Storage for local variables and parameters is allocated as needed during program execution. However, the compiler sets aside a chunk of memory (the *stack segment*), which is used for such allocations.

The size of the stack segment is fixed, and is determined at compile-time. The amount of the stack segment actually in use at any one time will depend on what procedures are suspended and what is currently executing.

Both the data and the stack segments are examples of *static memory allocation*. The term "static" is used because the amount of memory allocated is determined at compile-time (before the program executes) and also because the amount of memory is fixed, once determined.

Drawbacks of Static Memory Allocation

Sometimes, however, it may not be feasible to allocate storage statically. For example, suppose you need to write a program to rank numerical data. The program will get its data from the user, but the number of values to be sorted is not fixed.

For such a task, two approaches are possible:

- Declare an array with as many cells as possible, and accept at most that many values. If you allocate a large amount of storage but process only a few values, then your program is wasting memory that other parts of the program could perhaps use.
- Use a dynamic memory allocation strategy. Such a strategy involves pointers, as described in this chapter.

Dynamic Memory Allocation

In *dynamic memory allocation*, storage is not set aside until your program requests it while running. When storage is requested at run-time, the system allocates (and makes available to the program) the desired amount of contiguous storage.

This storage is allocated from a pool of available memory, known as the heap. The *heap* encompasses the area of memory that is still available after the program's code, data and stack segments have been loaded.

The Program Heap

The heap comprises whatever memory is allocated for use as needed. If your program is small and has little global data, and if there are no extraneous programs loaded into memory, then the heap can be quite large. The following program shows how to use a predefined function that provides information about the amount of available memory in the heap.

Listing 9.1

```
(* Program to illustrate use of MEMAVAIL function *)

PROGRAM HeapTest;
VAR DummyPtr : ^INTEGER;

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  NEW (DummyPtr);
   (* Determine how much memory is available
     in the entire heap --- less the storage for the integer
  *)
  WRITELN ( 'After allocating memory for an integer variable,' );
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 7, ' bytes; ');
  DISPOSE ( DummyPtr);
  WRITELN ( 'After returning memory allocated for the integer
variable,' );
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 7, ' bytes; ');
  AwaitUser;
END.
```

MEMAVAIL is a function that returns a **LONGINT** that represents the number of bytes of storage available in the heap. These bytes are not necessarily contiguous.

The heap is administered by a component of the CodeWarrior Pascal run-time library. This *Heap Manager* keeps track of available heap storage by maintaining a list that contains the status of all available storage in the heap.

Pointers

What happens when your program requests dynamic storage? To illustrate this process, you need to start working with pointers. A *pointer* is a data type that ``points to," or references, a particular location in memory. A pointer variable contains an address as its value. This address corresponds to a memory location. This memory location is available for use in your program, and is accessible through the pointer. The size of the memory location depends on the type of information to be stored there. Such memory is generally allocated dynamically, when your program requests it. The following program illustrates how to define and declare pointers, and how to get memory referenced by pointers.

Listing 9.2

```
(* Program to illustrate how to define and declare pointers,
   and how to get memory referenced by pointers.
*)
PROGRAM PtrDemo;

(* Define two pointers: pointer to REAL; pointer to INTEGER *)
TYPE IntPtr = ^INTEGER;
     RealPtr = ^REAL;

(* Declare two pointer variables *)
VAR IP : IntPtr;
    RP : RealPtr;
    ISize, RSize : INTEGER;

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  NEW ( IP);
  DISPOSE ( IP);
  (* Determine how much memory is available
     in the entire heap.
  *)
  WRITELN ( 'After allocating and deallocating ', SIZEOF ( IP^),
' bytes, ');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 7, ' bytes; ');

  (* Allocate storage for a REAL; RP will point to it *)
  NEW ( RP);

  (* RP^ is the area of memory to which RP points *)
  RSize := SIZEOF ( RP^);
  WRITE ( 'After allocating ', RSize : 7, ' bytes: ');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 7, ' bytes; ');

  (* Allocate storage for an INTEGER; IP will point to it *)
  NEW ( IP);
  ISize := SIZEOF ( IP^);
  WRITE ( 'After allocating ', ISize : 7, ' bytes: ');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 7, ' bytes; ');
  AwaitUser;
END.
```

Let's look at how pointers and their targets are used in this program. (We'll use *target* to refer to the memory location that corresponds to the value of a pointer. Thus, a pointer's contents represent the address of the pointer's target.)

Defining Pointer Types

Program **PtrDemo** contains two type definitions. The first,

```
IntPtr = ^INTEGER;
```

defines an **IntPtr** as a pointer to **INTEGER**. This means that an address stored in a variable of type **IntPtr** will be interpreted as the starting location of a two-byte chunk of memory. The contents of that chunk of memory will be interpreted as an **INTEGER**.

Similarly, the program defines **RealPtr** as a pointer to **REAL**. This means that an address stored in a **RealPtr** variable will be interpreted as the starting address of a four-byte chunk, whose contents will be interpreted as a **REAL**.

The caret (^) symbol before the base type specifier indicates that you're defining a pointer and not just a synonym for an existing type. The base type can be a simple or a structured type; in fact, it can even be a pointer type.

When defining a pointer type, you always need to specify the base type of the pointer's target --- so the program will know how much memory it should access at a pointer's target location.

Declaring Pointer Variables

Once you've specified what kinds of pointers your program will use, you can declare variables of these pointer types. Such declarations have the same format as any other declarations.

In the program, **IP** and **RP** are declared as variables of type **IntPtr** and **RealPtr**, respectively. Since they are declared in the main program, these are global variables. This means that the storage allocated for them is in the data segment. Thus, the storage for the pointer variables **IP** and **RP** is statically allocated. The storage corresponding to their targets, however, will be dynamically allocated.

TIP: A pointer variable is generally a static variable. The pointer's target, on the other hand, will usually be allocated dynamically.

Allocating Dynamic Storage

The calls to **NEW** in **PtrDemo1** allocate storage at run-time. The call

```
NEW ( RP);
```

is a request for dynamic memory allocation. The argument passed to **NEW** must be a pointer variable. The base type of the argument's target variable will determine the amount of storage being requested. Any storage returned will be on the heap.

When **NEW** is called with **RP** as its argument:

- The program asks the Heap Manager for enough contiguous memory (four bytes) to store a **REAL**.
- The Heap Manager finds such a chunk of memory in the heap. (This chunk of the heap is removed from a list of available memory locations that the Heap Manager keeps.)
- The location of this chunk of memory is stored in **RP**. (Thus, **RP** is assigned a value when **NEW** is called.)

Prior to the call to **NEW**, RP was uninitialized. With the call to **NEW**, RP gets a value. After this has all been accomplished, there also should be fewer available bytes on the heap. Run the program again. You'll find that the heap size has decreased. The heap size will actually shrink by more than the number of bytes needed to store the target variable, because the Heap Manager allocates heap memory in fixed size chunks.

The heap size decreases further when you allocate storage for **IP**'s target --- by calling **NEW** with **IP** as the argument. If the Heap Manager cannot allocate the desired amount of storage, the **NEW** routine will return a null pointer (a pointer whose address is the predefined value **NIL**), which essentially means that the pointer is not pointing to any usable storage. If you try to store something in the pointer's target variable, the program may crash with a run-time error.

One way to avoid such an error is to check whether there is enough storage available before trying to allocate the storage. The following code excerpt shows an example of how to do this:

```
(* Assume the following declarations:

   TYPE RealArray = ARRAY [ 1 .. 500] OF REAL;
        RAPtr = ^RealArray;
   VAR RAP : RAPtr;
*)

...
IF MEMAVAIL > SIZEOF ( RealArray) THEN
  ...
```

In order to be able to allocate storage for a new dynamic variable, the largest chunk of available heap must be at least as large as the storage required for a **RealArray** in the example. If this is not the case --- that is, if the heap has no more room --- the **IF** part of the loop is bypassed.

Clearly, if **MEMAVAIL** is less than the amount of storage needed for the target variable, then there is no point in trying to allocate storage. However, the available heap storage may be fragmented. If none of the available fragments is large enough, the allocation effort may still fail. Therefore, it's important to check whether the pointer variable has a non-**NIL** value, as in the following code:

```
...
IF RPT <> NIL THEN
  ...
```

```
...
```

Accessing a Target Variable

The following statement shows how to refer to a pointer's target variable:

```
RSize := SIZEOF ( RP^);
```

RP^ is the target variable whose address is stored in **RP**. Essentially, **RP^** means, ``the variable referenced (or pointed to) by **RP**." The target variable has no name of its own; rather, it is identified in terms of the pointer which references it.

Such a reference --- through the pointer --- is known as an *indirect reference*. This is in contrast to a *direct reference*, in which you specify the variable by name. When using indirect reference, you essentially have a named variable between you and the storage you actually want to manipulate (that is, the target variable).

The preceding assignment statement says to do the following:

- Determine the amount of storage allocated for variable **RP^**.
- Assign this value to **RSize**.

Assigning Values to Target Variables

The following program shows how to assign values to target variables.

Listing 9.3

```
(* Program to illustrate how to assign values to
   target variables
*)
```

```
PROGRAM PtrDemo2;
USES Windows;

(* Define two pointers: pointer to REAL; pointer to INTEGER *)
TYPE IntPtr = ^INTEGER;
     RealPtr = ^REAL;

(* Declare two pointer variables *)
VAR IP : IntPtr;
    RP : RealPtr;
    RandSeed : INTEGER;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer seed for random number generator', RandSeed);
  SRAND ( RandSeed);

  NEW ( IP);
  DISPOSE (IP);
  WRITELN ( 'After grabbing and returning ', SIZEOF ( IP^), '
bytes:');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
   (* Allocate storage for a REAL; RP will point to it *)
  NEW ( RP);
   (* Allocate storage for an INTEGER; IP will point to it *)
  NEW ( IP);

  REPEAT
    (* Assign a random value to RP's target *)
    RP^ := RAND / MAXINT;
    (* Assign a random value between 0 and 499 to IP's target *)
    IP^ := RAND MOD 500;
    WRITELN ( 'RP^ = ', RP^ : 10 : 5, '; IP^ = ', IP^ : 5);
  UNTIL ( RP^ < 0.5) AND ( IP^ < 250);

  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  AwaitUser;
END.
```

The assignment statements in the **REPEAT** loop store numerical values at two locations on the heap. Note that you will assign **REAL** values to **RP^** and **INTEGER** values to **IP^**. The target variables are just variables of specific types. Unlike ordinary variables, however, the dynamically allocated variables can only be accessed indirectly --- that is, through their pointers.

Don't Confuse Pointer and Target

You must be clear on what is stored where when you're working with pointers. You can only assign addresses to pointer variables. A pointer variable, such as **RP** in **PtrDemo2**, cannot be assigned an ordinary numerical value. Thus, the following assignment is invalid:

```
(* Assume RP is a RealPtr --- that is, ^REAL *)

(* invalid call, since RP is not of type REAL *)
RP := 23.5;
```

RP is *not* of type **REAL**; rather, it is a *pointer to* **REAL**.

It's just as important to avoid mistaking **RP^** for a pointer variable. For example, you can't do something like the following:

```
NEW ( RP^);
```

NEW requires a pointer argument, whereas **RP^** is of type **REAL**.

Pointer Assignments

You can assign the contents of one pointer variable to another pointer variable if both pointers are of the same type --- that is, both pointer must point to the same type of value. The following program shows how to assign the contents of one pointer variable to another:

Listing 9.4

```
(* Program to illustrate how to assign values to
   pointer variables
*)

PROGRAM PtrDemo3;

(* Define two pointer types: pointer to REAL and to INTEGER *)
TYPE IntPtr = ^INTEGER;
     RealPtr = ^REAL;

(* Declare four pointer variables *)
VAR IP, IP2 : IntPtr;
    RP, RP2 : RealPtr;

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
```

```
  NEW ( IP);
  DISPOSE (IP);
  WRITELN ( 'After grabbing and returning ', SIZEOF ( IP^), '
bytes:');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  (* Allocate storage for a REAL; RP will point to it *)
  NEW ( RP);
  (* Assign 23.5 to RP's target variable *)
  RP^ := 23.5;

  (* Assign the address stored in RP to RP2.
     After that, both will have the same target variable.
     This target variable will have 2 names: RP^ and RP2^
  *)
  RP2 := RP;
  WRITELN ( 'RP^ = ', RP^ : 10 : 5, '; RP2^ = ', RP2^ : 10 : 5);

  (* Allocate storage for an INTEGER; IP will point to it *)
  NEW ( IP);
  (* Assign 23 to IP's target variable *)
  IP^ := 23;
  WRITELN ( 'RP^ = ', RP^ : 10 : 5, '; IP^ = ', IP^ : 5);

  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  AwaitUser;
END.
```

Remember that the pointer variables are stored in the data or stack segment (depending on whether these variables are global or local); the target variables are stored on the heap. Although there are four pointers in this program, there are only two target values.

Once both pointers have the same value, you'll have two names for the same area of memory: **RP^** and **RP2^**. This is known as *aliasing*, and can cause problems if you're not careful.

Because a particular area of memory has multiple names associated with it, your program has various ways to change the value stored at this location. This makes it more difficult to trace errors that might involve the value stored at such a location.

For example, run the following program. Enter a large value, and then enter a small value (between 0 and 1). Look at the source code to see how **RP^** is changed.

Listing 9.5

```
(* Program to illustrate how to change a value by using
   an alias.
*)

PROGRAM AliasDemo;

TYPE RealPtr = ^REAL;
VAR RP, RP2 : RealPtr;
    ValCount : INTEGER;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

   (* Change ThePtr's target value:
      small values (between 0 and 1) become large;
      other values become small.
  *)
  PROCEDURE Transform ( VAR ThePtr : RealPtr);
  CONST MultFactor = 1000.0;
  BEGIN
    IF ( ThePtr^ >= 0) AND ( ThePtr^ < 1.0) THEN
      ThePtr^ := MultFactor * ThePtr^
    ELSE
      ThePtr^ := 1 / ThePtr^;
  END; (* Transform *)

BEGIN
  ValCount := 0;
  (* Allocate storage for a REAL; RP will point to it *)
  NEW ( RP);
  WRITELN ( 'After allocating ', SIZEOF ( RP^), ' bytes:');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  REPEAT
    GetReal ( 'Value? (< 0 to stop)', RP^);
    INC ( ValCount);
    (* Make RP2 point to the same location, so that this
       location now has two names: RP^ and RP2^
    *)
    RP2 := RP;
    WRITELN ( 'RP^ = ', RP^ : 10 : 5);

    (* Change a target value;
       note that no mention is made of RP.
    *)
    Transform ( RP2);

    (* Although it was never used explicitly,
       RP^ has a new value.
    *)
    WRITELN ( 'After Transform:');
    WRITELN ( 'RP^ = ', RP^ : 10 : 5);
  UNTIL ( RP^ < 0);
  WRITELN ( 'After reading ',  ValCount, ' values:');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  AwaitUser;
END.
```

This program essentially converts large values to small ones, and vice versa. The user enters a value, which is stored at **RP^**. Although **RP^** is used only in the call to **GetString** and in two calls to **WRITELN**, its value is nonetheless changed. What makes the program interesting is the way in which this is accomplished.

After assigning a value to **RP^**, the program makes a second pointer refer to the same memory location, by assigning the address stored in **RP** to **RP2**. At this point, both **RP** and **RP2** contain the same address.

The call to **Transform** involves only the pointer **RP2** explicitly. No mention is made of a target variable. However, keep in mind that a pointer almost always has a target variable associated with it. If you can determine the pointer's contents (to find out the address stored there), you can change the value stored at the pointer's target.

In this case, however, the location at which the target is stored also has another name in the main program: **RP^**. Thus, when you change **ThePtr^** in **Transform**, you're also changing **RP^**.

Notice that **Transform** has a **VAR** parameter. This was done to raise a point. The procedure doesn't really change the value of **RP2**. Therefore, this could be a value parameter, and you would still get the same results. To convince yourself, remove the **VAR** from the interface for **Transform**, and then run the program again. For the same input values, you'll get the same results.

To see why this is the case, let's work with a simplified concept of an address. Assume that memory addresses are simply integer values within a given range.

Suppose that the address stored in **RP2** (and **RP**) is 5000. Now suppose the program calls **Transform**, and passes **RP2** by *value*. In that case, the contents of **RP2** are copied and passed into the procedure.

Within **Transform**, the value 5000 would still be interpreted as an address. Thus, **ThePtr^** would be interpreted as the value stored at location 5000. If this value is changed, it will still be changed when the procedure finishes executing. The value stored at **RP2** is not accessible, even if the target associated with this value is. Thus, when you pass a pointer by value, you can change the contents of the pointer's target, but not of the pointer itself.

Assigning Addresses to Pointers

So far, you have initialized pointers by calling **NEW** or by assigning the contents of another pointer variable to the pointer. CodeWarrior Pascal also allows you to assign the address of a specific variable to a pointer --- even if that variable is statically defined.

The Address, or Pointer, Operator

CodeWarrior Pascal provides an operator for determining an address. The *address*, or *pointer*, operator (@) produces the location of its argument, which must be a variable, a parameter, or a routine. Thus, @ takes an argument that has a memory location, and returns the location of this object. @ is a unary operator, and has the same precedence as **NOT**, which is the highest precedence.

WARNING! The at sign (@) used for the address operator is used in some older Pascal implementations as an alternative to the caret (^) for specifying a pointer. You'll need to make changes if you use this operator in CodeWarrior Pascal, and you then move your programs to different Pascal environments.

The following program shows how to use @ to get an address:

Listing 9.6

```
(* Program to illustrate use of @ for specifying
   an address.
*)

PROGRAM AddressOpDemo;

TYPE RealPtr = ^REAL;
VAR RP1, RP2 : RealPtr;
    RVal1, RVal2 : REAL;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  RVal1 := 100.0;
  RVal2 := 200.0;

   (* Assign the address of RVal1 to RP1 *)
  RP1 := @RVal1;
   (* Assign the address of RVal1 to RP2 *)
  RP2 := @RVal1;

  IF RP1 = RP2 THEN
    WRITELN ( 'Pointers contain the same address')
  ELSE
    WRITELN ( 'Pointers contain DIFFERENT addresses');

  WRITELN ( 'RP1^ = ', RP1^ : 10 : 5,
            '; RP2^ = ', RP2^ : 10 : 5);
  AwaitUser;
END.
```

Pointer Pitfalls

Pointers are very powerful programming aids. This power has its price, however, and you can get yourself into lots of trouble with pointers --- if you're not careful. In this section, we'll look at a common problem that can arise when using pointers.

Reusing a Pointer

As you've seen, pointers generally get their target variables from the program's heap --- through dynamic memory allocation. But what happens if you reuse a pointer already associated with a memory location? For example, what happens if you call **NEW** multiple times, with the same pointer as argument each time? The following program will show you:

Listing 9.7

```
(* Program to illustrate how to reuse a pointer,
   and also what happens when you do so.
*)

PROGRAM ReUseDemo;
USES Windows;
```

```
CONST MaxTimes = 20;
TYPE RealPtr = ^REAL;
VAR RP : RealPtr;
    Count, RandSeed : INTEGER;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Allocate storage for a REAL, generate a random value
      and assign the value to newly allocated storage.
      NOTE that thei procedure has a VAR parameter.
  *)
  PROCEDURE GetVal ( VAR ThePtr : RealPtr;
                     Count : INTEGER);
  BEGIN
    NEW ( ThePtr);
    ThePtr^ := RAND / MAXINT;
    WRITELN ( Count : 2, ': ThePtr^ = ', ThePtr^ : 10 : 5);
  END; (* GetVal *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer seed for random number generator', RandSeed);
  SRAND ( RandSeed);
  NEW (RP);
  DISPOSE ( RP);
  WRITELN ( 'After allocating and returning ', SIZEOF ( RP^), '
bytes:');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  FOR Count := 1 TO MaxTimes DO
    GetVal ( RP, Count);
  WRITELN ( '    RP^     = ', RP^ : 10 : 5);
  (* Note that MaxTimes * SIZEOF (REAL) bytes of storage have
been used *)
  WRITELN ( 'After getting memory for ', MaxTimes, ' values:');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  AwaitUser;
END.
```

This program calls a procedure multiple times. Each time, this procedure allocates storage for a **REAL**, and makes **ThePtr** reference this variable. (Here, **ThePtr** corresponds to the global **RP**.) After doing this the required number of times, the program checks on the available heap storage.

Notice that the program has lost over 400 bytes of heap. At the end of the program, however, there is only one **RP^**. The other 19 have been lost. With each call to **NEW**, the contents of **ThePtr** were changed. By writing over an address, the program loses access to the memory location stored at that address. This inability to reuse pointers is a severe handicap.

Fortunately, there is a very elegant way around this obstacle. The solution enables you to create some very powerful data structures, such as linked lists. You'll learn about such methods later in the section ``Linking Self-Referential Structures."

Returning Storage to the Heap

It's possible to return storage to the heap. This process is also known as *deallocation*, and makes the storage available for reuse. The predefined **DISPOSE** procedure enables you to return storage that had been allocated using **NEW**.

The procedure takes a pointer as its parameter, and returns the storage allocated for the pointer's target variable to the heap for reuse. Run the following program, and compare its output with that of **ReUseDemo**.

Listing 9.8

```
(* Program to illustrate how to return storage to the
   heap for reuse.
*)

PROGRAM ReturnDemo;
USES Windows;

CONST MaxTimes = 20;
TYPE RealPtr = ^REAL;
VAR RP : RealPtr;
    Count, RandSeed : INTEGER;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Allocate storage for a REAL, generate a random value
      and assign the value to newly allocated storage.
      NOTE that this procedure has a VAR parameter.
      NOTE also that the procedure first gives back some storage.
  *)
  PROCEDURE GetVal ( VAR ThePtr : RealPtr;
                     Count : INTEGER);
  BEGIN
    DISPOSE ( ThePtr);
    NEW ( ThePtr);
    ThePtr^ := RAND / MAXINT;
    WRITELN ( Count : 2, ': ThePtr^ = ', ThePtr^ : 10 : 5);
  END; (* GetVal *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
```

```
  GetInt ( 'Integer seed for random number generator', RandSeed);
  SRAND ( RandSeed);
  (* Necessary because GetVal calls DISPOSE *)
  NEW ( RP);
  WRITELN ( 'After allocating memory for the first value, ');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  FOR Count := 1 TO MaxTimes DO
    GetVal ( RP, Count);
  WRITELN ( '    RP^     = ', RP^ : 10 : 5);
  (* Note that only SIZEOF ( REAL) bytes of storage have been
used *)
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  AwaitUser;
END.
```

The major difference between the two programs is in the amount of heap storage used. The second version loses less, because the pointer's target variable is returned to the heap before a new location is stored in **ThePtr**.

DISPOSE causes a run-time error if its pointer argument has no associated target variable. The call to **NEW** in the main program is needed to avoid such an error. You can only use **DISPOSE** with pointers that have been assigned values with **NEW**.

*NIL: **A Predefined Value for Pointers***

There's a way to avoid calling **DISPOSE** when its pointer argument is undefined. You can use a special address, **NIL**, that is predefined for use with pointers. **NIL** represents a special location that is assigned to pointers to give them an initial value.

A pointer whose value is **NIL** does not have a target. However, the contents indicate that the pointer has been initialized --- a fact you can use to control program execution. The following program accomplishes the same thing as **ReturnDemo**, but does not call **NEW** in the main program. Instead, the program initializes **RP** to **NIL**.

Listing 9.9

```
(* Program to illustrate how to return storage to the
   heap for reuse.
*)

PROGRAM NILDemo;
USES Windows;

CONST MaxTimes = 20;
TYPE RealPtr = ^REAL;
VAR RP : RealPtr;
    Count, RandSeed : INTEGER;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Allocate storage for a REAL, generate a random value
      and assign the value to newly allocated storage.
      NOTE that this procedure has a VAR parameter.
      NOTE also that the procedure first gives back some storage.
  *)
  PROCEDURE GetVal ( VAR ThePtr : RealPtr;
                     Count : INTEGER);
  BEGIN
    IF ThePtr <> NIL THEN
      DISPOSE ( ThePtr);
    NEW ( ThePtr);
    ThePtr^ := RAND / MAXINT;
    WRITELN ( Count : 2, ': ThePtr^ = ', ThePtr^ : 10 : 5);
  END; (* GetVal *)

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
```

```
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetInt *)

BEGIN
  GetInt ( 'Integer seed for random number generator', RandSeed);
  SRAND ( RandSeed);
  NEW ( RP);
  DISPOSE ( RP);
  WRITELN ( 'After allocating and returning memory for a REAL,
');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  (* Necessary because GetVal calls DISPOSE *)
  RP := NIL;
  FOR Count := 1 TO MaxTimes DO
    GetVal ( RP, Count);
  WRITELN ( '    RP^     = ', RP^ : 10 : 5);
  (* Note that MaxTimes * SIZEOF ( REAL) bytes of storage have
been used *)
  WRITELN ( 'After getting ', MaxTimes, ' values, ');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  AwaitUser;
END.
```

In this version, **GetVal** simply tests whether **ThePtr** has been initialized to **NIL**. If so, **DISPOSE** is *not* called. It's always a good idea to initialize any pointers you have by assigning **NIL** to them, as in **NILDemo**.

Self-Referential Structures

From the examples you've seen so far, you may well get the impression that pointers are more trouble than they're worth. In this section, you'll learn about a type of task for which dynamic memory allocation and the use of pointers are extremely well suited.

Often, a program will use multiple elements of the same type, with the number of elements undetermined until run-time. For example, suppose you want to alphabetize the words read from a file, but you don't know the number of words in advance. You should use a dynamic storage allocation strategy for such a task --- to allocate storage for each element as necessary.

The basic data elements in such a program will almost always be some sort of record. In Chapter 7, you learned that a record cannot have itself as a field, but that it *can* have a pointer to another record like itself as a field. It's time to elaborate on that statement. Consider the following excerpt from a declaration section:

```
TYPE LNodePtr = ^LNode;
     LNode = RECORD
               Info : REAL;
               Next : LNodePtr;
             END; (* LNode RECORD *)

VAR Root : LNodePtr;
```

This listing describes a record called an **LNode**. This record has two fields. The **Info** field is used to store an item of numerical information. The **Next** field is a pointer to another element of the same type --- that is, to another **LNode**. Thus, an **LNode** is a special kind of structure. It not only contains numerical information, but also contains the information needed to access another **LNode**.

Linking Self-Referential Structures

If each **LNode** can reference, or point to, another **LNode**, then it should be clear that you can build arbitrarily long chains of such structures, linked to one another by their **Next** fields. A *linked list* is a chain of elements, where each element is linked to the next one. (There are other types of chains, such as *trees*.) The individual elements in such a chain are often called *nodes*. The term node is a general one that refers to components in various types of sequences. In the example programs, **LNode** specifies a node for a linked list.

The first **LNode** in a linked list is often referenced by a pointer to **LNode** --- that is, by an ordinary pointer, *not* an **LNode** or **LNode** field. Thus, the **Root** variable declared in the previous listing is an **LNodePtr**, not an **LNode**. Root will be set to point to the first node in a list.

The **Next** field of the first **LNode** may, in turn, reference another **LNode**. The list may contain as many **LNode**s as your program can fit in the available heap memory. The value of the **Next** field of the last **LNode** in the list should be **NIL**, to indicate that there are no more nodes.

Building Linked Lists

A linked list is a dynamic data structure in that its size is not determined until run time. This is different from an array, for which you specify the size and the base type in advance.

You must build a linked list node by node. To add a node to an existing list:

- Get storage for the node via a call to **NEW** with the appropriate argument.
- Store your information in the **LNode**.
- Connect the new node to the existing linked list, by making the appropriate pointer from the linked list reference your new node. During this process the program may also make the **Next** field of the new **LNode** point to an **LNode** already in the list.

Adding at the Front of a Linked List

You can build a linked list in which each new node is added at the front of the list, as shown in the following program:

Listing 9.10

```
(* Program for building a linked list,
   by adding elements to the FRONT of the list.
*)

PROGRAM LListDemo;

TYPE LNodePtr = ^LNode;

     LNode = RECORD
               Info : REAL;
               (* a pointer to another record of this type *)
               Next : LNodePtr;
             END;

VAR Root, Temp : LNodePtr;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

  (* Write the info from each node in a linked list *)
  PROCEDURE WriteList ( VAR TheNode : LNodePtr);
  BEGIN
    WRITELN ( TheNode^.Info : 10 : 5);
    IF ( TheNode^.Next <> NIL) THEN
      WriteList ( TheNode^.Next);
  END; (* WriteList *)

  (* get space for and initialize a linked list node *)
  PROCEDURE InitLNode ( VAR TheNode : LNodePtr);
  BEGIN
    NEW ( TheNode);
    TheNode^.Info := 0.0;
    TheNode^.Next := NIL;
  END; (* InitLNode *)

  (* insert new node (ToAdd) at the front
     of a linked list (WhereToAdd)
  *)
  PROCEDURE LLAtFront ( ToAdd : LNodePtr;
                        VAR WheretoAdd : LNodePtr);
  BEGIN
    ToAdd^.Next := WhereToAdd;
    WhereToAdd := ToAdd;
  END; (* LLAtFront  *)

BEGIN (* Main *)
```

```
  (* Initialize the pointers *)
  Root := NIL;
  Temp := NIL;
  REPEAT
    (* Get storage for an LNode, and initialize it *)
    InitLNode ( Temp);
    GetReal ( 'Value (< 0 to stop)', Temp^.Info);
    (* Add new LNode to the front of the existing list *)
    LLAtFront ( Temp, Root);
  UNTIL Temp^.Info < 0.0;
  (* Display contents of the linked list, in order *)
  WriteList ( Root);
  AwaitUser;
END.
```

The program works with **Root** and **Temp**, two pointers to **LNode**. Root serves as the permanent anchor for the linked list that will be built. Thus, **Root** always points to the first node in the list, although the first node changes as the program executes. You need to go through **Root** to access any element in the list.

Temp ``holds" new nodes while they are being initialized and before they are added to the list. The general program strategy is to make **Temp** point to the storage for each new node, and then to attach this node to the list. **Temp** is then free to reference the next new **LNode**. By attaching the **LNode** to the list, you keep the storage accessible while freeing **Temp** for another **LNode**. Note that this use of **Temp** actually makes use of aliasing.

InitLNode

Procedure **InitLNode** gets the storage required for a new **LNode**, and returns a pointer to this storage. As in earlier examples, **NEW** is called for the storage. The argument to **NEW** says that you want enough new space to store an **LNode**. If the program is unable to allocate storage, then a run-time error occurs.

Make sure the **Next** field of every new node is initialized to a **NIL** pointer (that is, a pointer whose value is **NIL**). Many list handling routines look for such a value to determine whether they have reached the end of a list. If you don't initialize these pointers, the last node may not contain a **NIL** pointer, even though it should. The program may try to read past the end of the list in such a case, thereby causing the program to crash.

Notice the syntax for specifying a field in a target variable that also happens to be a record:

TheNode^.Info, rather than **TheNode.Info^**. The syntax for specifying a target variable is to put the variable's identifier followed by a caret (^). In procedure **InitLNode**, the variable's name is **TheNode**. Once you've specified the target variable, you can access individual fields by putting a period followed by the field name.

LLAtFront

After the storage is allocated, the **Info** field of the **LNode** is assigned a value. Then procedure **LLAtFront** is called to insert the new node at the beginning of the linked list.

The basic strategy is first to make the **Next** field of **ToAdd^** reference the current first node. Then the **WhereToAdd** pointer (that is, **Root**) is made to point to the new node.

The order in which these steps are carried out is important. Recall that the only way to access nodes in a list is through the pointer that references the first node. This is because the list anchor is the only statically allocated variable in the linked list and is, thus, the only variable in the list that is directly accessible.

If you first made **WhereToAdd** point to the new node, you would lose access to the rest of the existing list, since the assignment would remove the only available access to the first element in the original list (**WhereToAdd^**).

WriteList

This procedure displays the contents of the individual nodes in the linked list, in the order in which they are encountered by the routine. Notice that the procedure is recursive.

The procedure does two things: display the **Info** field of the current **LNode** and call a version of itself to handle the next **LNode** in the list. Eventually, the recursive call will be unnecessary because the routine will reach the last node in the list --- the one whose **Next** field has the value **NIL**.

Because the instruction to display the **Info** field precedes the recursive call, the nodes are written from first to last. By moving the **WRITELN** call to follow the recursive call, you can have **WriteList** write the nodes from last to first.

TIP: To access any element in a linked list, you need to start with the pointer to the first element in the list (Root, in the main program of several examples).

Adding Elsewhere in a Linked List

You can also build a linked list by adding each new element to the end of the current list, as in the following procedure. To see how **LLAtBack** works, add the procedure definition to the declaration section of the **LListDemo** program. You'll also need to change the call from **LLAtFront** to **LLAtBack** in the main program.

```
(* insert a new node (ToAdd) at the back
   of an existing linked list (WhereToAdd)
*)
PROCEDURE LLAtBack ( ToAdd : LNodePtr;
                     VAR WhereToAdd : LNodePtr);
BEGIN
  (* if at the end of the list, can add *)
  IF ( WhereToAdd = NIL) THEN
    WhereToAdd := ToAdd
  ELSE (* if not at the end, keep looking *)
    LLAtBack ( ToAdd, WhereToAdd^.Next);
END; (* LLAtBack *)
```

This procedure adds the new node to the end of the current list. The strategy is to *traverse*, or move through, the list until you reach the end of the list. You can recognize this point because you'll find a pointer with **NIL** as its value. (This is guaranteed because **InitLNode** initializes the **Next** field to **NIL**.)

If the procedure reaches a point in the list with aa **NIL** value, then the new node is simply added and the modified list is passed back to the calling routine. If there is a node at the current position, then the **LLAtBack** procedure calls a version of itself with the **Next** field of the current node (**WhereToAdd^.Next**) as the new argument. When the end of the list is finally reached, the last call to **LLAtBack** will return a list containing the new node to its calling routine.

Thus, this outer list will be changed, and will pass on this change to its calling routine. At the outermost part of this calling sequence, the list referenced by **Root** will have been changed, which is exactly what you want.

The following program lets you add to the end of a list.

Listing 9.11

```
(* Program for building a linked list,
   by adding elements to the BACK of the list.
*)
```

```
PROGRAM LListDemo2;

TYPE LNodePtr = ^LNode;

     LNode = RECORD
               Info : REAL;
               (* a pointer to another record of this type *)
               Next : LNodePtr;
             END;

VAR Root, Temp : LNodePtr;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

   (* Write the info from each node in a linked list *)
  PROCEDURE WriteList ( VAR TheNode : LNodePtr);
  BEGIN
    WRITELN ( TheNode^.Info : 10 : 5);
    IF ( TheNode^.Next <> NIL) THEN
      WriteList ( TheNode^.Next);
  END; (* WriteList *)

   (* get space for and initialize a linked list node *)
  PROCEDURE InitLNode ( VAR TheNode : LNodePtr);
  BEGIN
    NEW ( TheNode);
    TheNode^.Info := 0.0;
    TheNode^.Next := NIL;
  END; (* InitLNode *)

  (* insert a new node (ToAdd) at the back
     of an existing linked list (WhereToAdd)
  *)
  PROCEDURE LLAtBack ( ToAdd : LNodePtr;
                       VAR WhereToAdd : LNodePtr);
  BEGIN
    (* if at the end of the list, can add *)
    IF ( WhereToAdd = NIL) THEN
      WhereToAdd := ToAdd
    ELSE (* if not at the end, keep looking *)
      LLAtBack ( ToAdd, WhereToAdd^.Next);
  END; (* LLAtBack *)

  (* insert new node (ToAdd) at the front
     of a linked list (WhereToAdd)
  *)
  PROCEDURE LLAtFront ( ToAdd : LNodePtr;
                        VAR WheretoAdd : LNodePtr);
  BEGIN
    ToAdd^.Next := WhereToAdd;
    WhereToAdd := ToAdd;
  END; (* LLAtFront  *)

BEGIN (* Main *)

  (* Initialize the pointers *)
  Root := NIL;
  Temp := NIL;
  REPEAT
    (* Get storage for an LNode, and initialize it *)
    InitLNode ( Temp);
    GetReal ( 'Value (< 0 to stop)', Temp^.Info);
    (* Add new LNode to the front of the existing list *)
    LLAtBack ( Temp, Root);
  UNTIL Temp^.Info < 0.0;
  (* Display contents of the linked list, in order *)
  WriteList ( Root);
  AwaitUser;
END.
```

Note that the only changes we had to make to the previous program were:

- Adding Procedure **LLAtBack**, and
- Changing the call from **LLAtFront** to **LLAtBack**.
- Changing the comments.

Ordering a Linked List

Sometimes you'll want to put the elements in your list into some type of order. For example, you might want to order the nodes on the basis of numerical value. The following procedure lets you build such an ordered list, putting the node with the lowest value for **Info** at the front of the list. To see how the procedure works, add its source code to **LListDemo** and change the call in the main program from **LLAtBack** to **LLInMiddle**.

```
(* insert new node (ToAdd) in lexicographic position in a
   linked list (WhereToAdd)
*)
PROCEDURE LLInMiddle ( ToAdd : LNodePtr;
                       VAR WhereToAdd : LNodePtr);
BEGIN
  (* if at the end, (and not yet added,) add the node *)
  IF ( WhereToAdd = NIL) THEN
    WhereToAdd := ToAdd
  (* if the node should be added at the current spot *)
  ELSE IF ( WhereToAdd^.Info >= ToAdd^.Info) THEN
  BEGIN
    ToAdd^.Next := WhereToAdd;
    WhereToAdd := ToAdd;
  END
  ELSE (* if still not able to add node *)
    LLInMiddle ( ToAdd, WhereToAdd^.Next);
END; (* LLInMiddle *)
```

This procedure shares characteristics with each of the other two list building routines. The basic strategy is to move through the list, comparing the appropriate values of the new node and of the current node in the list. (The current node is the one referenced by the **WhereToAdd** parameter in the function, that is, **WhereToAdd^**).

If the list is empty or the end of the list has been reached, then the new node is simply added --- just as in **LLAtBack**. If the list value is larger than the value in the current node, then the new node is added before the current node in the list. This process is identical to adding an element at the front of a list. In fact, you are actually adding the new node to the front of a sublist containing only those nodes with values greater than the value in the new node. (What corresponds to **Root** in this case?)

If the list value is still smaller than the value in the new node, then the program moves to the next node in the list. This should have a larger value, since the list is ordered after each node is added.

Eventually, the new node is added to a list, and the modified list is passed back through the calling sequence, until the main program gets the modified list.

Example: Sorting Lines From A File

Suppose you want to read the contents of a file, and write them to another file --- but in alphabetic order. The following program shows how to do this using linked lists. Don't be intimidated --- much of the source code is auxiliary routines (such as **ShowProgress** and **FOpened**). The code involving linked lists is actually quite short.

Listing 9.12

```
(* Program for sorting lines read from a file,
   and writing them to a new file.
*)

PROGRAM SortLines;

TYPE RegStr = STRING[80];

     LNodePtr = ^LNode;
     LNode = RECORD
               Info : RegStr;
               (* a pointer to another record of this type *)
               Next : LNodePtr;
             END;

VAR Root, Temp : LNodePtr;
```

```
    Source, Log : Text;
    SName, LName : STRING;
    NrLines : INTEGER;

(* *****************                    *************** *)
(* ***************** Auxiliary Routines *************** *)
(* *****************                    *************** *)

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Open an existing file, doing all necessary
     and warranted checking, and taking all
     precautions.
     Returns TRUE if file was opened; FALSE otherwise.
  *)
  FUNCTION FOpened ( VAR TheFile : Text;
                     FName : STRING) : BOOLEAN;
  BEGIN
  ASSIGN ( TheFile, FName);
  (*$I-*)
  RESET ( TheFile);
  (*$I+*)
  IF IOResult = 0 THEN
    FOpened := TRUE
  ELSE
    FOpened := FALSE;
  END; (* FN FOpened *)

  (* Create a new file, doing all necessary
     and warranted checking, and taking all
     precautions.
     Returns TRUE if file was created; FALSE otherwise.
  *)
  FUNCTION FMade ( VAR TheFile : Text;
                  FName : STRING) : BOOLEAN;
  BEGIN
    ASSIGN ( TheFile, FName);
    (*$I-*)
    REWRITE ( TheFile);
    (*$I+*)
    IF IOResult = 0 THEN
      FMade := TRUE
    ELSE
      FMade := FALSE;
  END; (* FN FMade *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

  (* Write '.' periodically, to let the user know that
     the program is doing something.
     Val is the counter being tested;
     Small is the cycle for a '.'
     Large is the cycle for a WRITELN
  *)
  PROCEDURE ShowProgress ( Val, Small, Large : LONGINT);
  CONST MarkerCh = '.';
  BEGIN
    IF ( Val MOD Small = ( Small - 1)) THEN
      WRITE ( MarkerCh);
    IF ( Val MOD Large = ( Large - 1)) THEN
      WRITELN;
  END; (* ShowProgress *)

(* *****************                       *************** *)
(* ***************** Linked List Routines *************** *)
(* *****************                       *************** *)

  (* get space for and initialize a linked list node *)
  PROCEDURE InitLNode ( VAR TheNode : LNodePtr);
  BEGIN
    NEW ( TheNode);
    TheNode^.Info := '';
    TheNode^.Next := NIL;
  END; (* InitLNode *)

  (* insert new node (ToAdd) in lexicographic position in a
     linked list (WhereToAdd)
  *)
  PROCEDURE LLInMiddle ( ToAdd : LNodePtr;
                         VAR WhereToAdd : LNodePtr);
  BEGIN
    (* if at the end, (and not yet added,) add the node *)
    IF ( WhereToAdd = NIL) THEN
      WhereToAdd := ToAdd
    ELSE (* if the node should be added at the current spot *)
      IF ( WhereToAdd^.Info >= ToAdd^.Info) THEN
        BEGIN
          ToAdd^.Next := WhereToAdd;
          WhereToAdd := ToAdd;
        END
      ELSE (* if still not able to add node *)
        LLInMiddle ( ToAdd, WhereToAdd^.Next);
  END; (* LLInMiddle *)

  (* Write the info from each node in a linked list.
     Note that this is an iterative routine.
  *)
  PROCEDURE IterFWriteList ( VAR TheFile : Text;
                             TheNode : LNodePtr);
  BEGIN
    (* Write as long as there's still something to write *)
    WHILE TheNode <> NIL DO
    BEGIN
      WRITELN ( TheFile, TheNode^.Info);
      TheNode := TheNode^.Next;
    END;
  END; (* IterFWriteList *)

BEGIN (* Main *)
```

```
  (* Initialize the pointers *)
  NEW ( Root);
  DISPOSE ( Root);
  Root := NIL;
  Temp := NIL;
  NrLines := 0;
  WRITELN ( 'Before processing any lines, ');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  GetString ( 'Source file?', SName);
  GetString ( 'Log file?', LName);

  IF FOpened ( Source, SName) AND FMade ( Log, LName) THEN
  BEGIN
    WRITELN ( SName, ' opened and ', LName, ' created
successfully');
    WRITELN ( 'Getting ready to process file contents');
    WHILE ( NOT EOF ( Source)) AND
          ( MEMAVAIL > SIZEOF ( LNode)) DO
    BEGIN
      (* Get storage for an LNode, and initialize it *)
      InitLNode ( Temp);
      READLN ( Source, Temp^.Info);
      INC ( NrLines);
      ShowProgress ( NrLines, 5, 250);
      (* Add new line to linked list, which is always sorted *)
      LLInMiddle ( Temp, Root);
    END;  (* WHILE NOT EOF and still heap available *)
    IF NOT EOF ( Source) THEN
      WRITELN ( 'Terminated early; not enough memory.');
    (* Display contents of the linked list, in order *)
    IterFWriteList ( Log, Root);
    CLOSE ( Log);
    CLOSE ( Source);
    WRITELN;
    WRITELN ( NrLines : 5, ' lines processed');
    WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  END  (* IF FOpened and FMade *)
  ELSE
    WRITELN ( 'File error.');
  AwaitUser;
END.
```

```
(* Program for sorting lines read from a file,
   and writing them to a new file.
*)

PROGRAM SortLines;

TYPE RegStr = STRING[80];

     LNodePtr = ^LNode;
     LNode = RECORD
               Info : RegStr;
               (* a pointer to another record of this type *)
               Next : LNodePtr;
             END;

VAR Root, Temp : LNodePtr;
    Source, Log : Text;
    SName, LName : STRING;
    NrLines : INTEGER;

(* *****************                    *************** *)
(* ***************** Auxiliary Routines *************** *)
(* *****************                    *************** *)

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Open an existing file, doing all necessary
     and warranted checking, and taking all
     precautions.
   Returns TRUE if file was opened; FALSE otherwise.
*)
FUNCTION FOpened ( VAR TheFile : Text;
                   FName : STRING) : BOOLEAN;
BEGIN
ASSIGN ( TheFile, FName);
(*$I-*)
RESET ( TheFile);
(*$I+*)
IF IOResult = 0 THEN
  FOpened := TRUE
ELSE
  FOpened := FALSE;
END; (* FN FOpened *)

(* Create a new file, doing all necessary
   and warranted checking, and taking all
   precautions.
   Returns TRUE if file was created; FALSE otherwise.
*)
FUNCTION FMade ( VAR TheFile : Text;
                 FName : STRING) : BOOLEAN;
BEGIN
  ASSIGN ( TheFile, FName);
  (*$I-*)
  REWRITE ( TheFile);
  (*$I+*)
  IF IOResult = 0 THEN
    FMade := TRUE
  ELSE
    FMade := FALSE;
END; (* FN FMade *)

PROCEDURE GetString ( Message : STRING;
                      VAR Value : STRING);
BEGIN
  WRITE ( Message, ' ');
  READLN ( Value);
END; (* GetString *)
```

```
  (* Write '.' periodically, to let the user know that
     the program is doing something.
     Val is the counter being tested;
     Small is the cycle for a '.'
     Large is the cycle for a WRITELN
  *)
  PROCEDURE ShowProgress ( Val, Small, Large : LONGINT);
  CONST MarkerCh = '.';
  BEGIN
    IF ( Val MOD Small = ( Small - 1)) THEN
      WRITE ( MarkerCh);
    IF ( Val MOD Large = ( Large - 1)) THEN
      WRITELN;
  END; (* ShowProgress *)

(* *****************                      *************** *)
(* ***************** Linked List Routines *************** *)
(* *****************                      *************** *)

  (* get space for and initialize a linked list node *)
  PROCEDURE InitLNode ( VAR TheNode : LNodePtr);
  BEGIN
    NEW ( TheNode);
    TheNode^.Info := '';
    TheNode^.Next := NIL;
  END; (* InitLNode *)

  (* insert new node (ToAdd) in lexicographic position in a
     linked list (WhereToAdd)
  *)
  PROCEDURE LLInMiddle ( ToAdd : LNodePtr;
                         VAR WhereToAdd : LNodePtr);
  BEGIN
    (* if at the end, (and not yet added,) add the node *)
    IF ( WhereToAdd = NIL) THEN
      WhereToAdd := ToAdd
    ELSE (* if the node should be added at the current spot *)
      IF ( WhereToAdd^.Info >= ToAdd^.Info) THEN
        BEGIN
          ToAdd^.Next := WhereToAdd;
          WhereToAdd := ToAdd;
        END
      ELSE (* if still not able to add node *)
        LLInMiddle ( ToAdd, WhereToAdd^.Next);
  END; (* LLInMiddle *)

  (* Write the info from each node in a linked list.
     Note that this is an iterative routine.
  *)
  PROCEDURE IterFWriteList ( VAR TheFile : Text;
                                 TheNode : LNodePtr);
  BEGIN
    (* Write as long as there's still something to write *)
    WHILE TheNode <> NIL DO
    BEGIN
      WRITELN ( TheFile, TheNode^.Info);
      TheNode := TheNode^.Next;
    END;
  END; (* IterFWriteList *)

BEGIN (* Main *)
  (* Initialize the pointers *)
  NEW ( Root);
  DISPOSE ( Root);
  Root := NIL;
  Temp := NIL;
  NrLines := 0;
  WRITELN ( 'Before processing any lines, ');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  GetString ( 'Source file?', SName);
  GetString ( 'Log file?', LName);

  IF FOpened ( Source, SName) AND FMade ( Log, LName) THEN
  BEGIN
    WRITELN ( SName, ' opened and ', LName, ' created 
successfully');
    WRITELN ( 'Getting ready to process file contents');
    WHILE ( NOT EOF ( Source)) AND
          ( MEMAVAIL > SIZEOF ( LNode)) DO
    BEGIN
```

```
      (* Get storage for an LNode, and initialize it *)
      InitLNode ( Temp);
      READLN ( Source, Temp^.Info);
      INC ( NrLines);
      ShowProgress ( NrLines, 5, 250);
      (* Add new line to linked list, which is always sorted *)
      LLInMiddle ( Temp, Root);
    END;  (* WHILE NOT EOF and still heap available *)
    IF NOT EOF ( Source) THEN
      WRITELN ( 'Terminated early; not enough memory.');
    (* Display contents of the linked list, in order *)
    IterFWriteList ( Log, Root);
    CLOSE ( Log);
    CLOSE ( Source);
    WRITELN;
    WRITELN ( NrLines : 5, ' lines processed');
    WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
  END  (* IF FOpened and FMade *)
  ELSE
    WRITELN ( 'File error.');
  AwaitUser;
END.
```

This program reads lines from **Source**, and adds them to a linked list. The nodes in the list are ordered on the basis of the strings (the **Info** fields) in each node. The program then writes the contents of this linked list, in order, to a log file.

Note that the definition of **LNode** in this program differs from the definition in earlier programs. This time, the **Info** field is a **STRING** type, rather than a **REAL**. In particular, this is a string with at most 80 characters. By using a limited string size, you can fit lots more nodes on the heap.

- The list handling routines are called in the main program's **WHILE** loop, and in the first statement after this loop. As you saw earlier, the program gets storage for each new line. This is accomplished using **InitLNode**, just as in earlier programs.

- The new storage is added to the existing list by procedure **LLInMiddle**. This routine is identical to the version in earlier programs. The power of a routine such as this one becomes clearer when you consider that the 8 or 9 lines that make up the procedure body are actually sorting a collection of strings.

- Finally, the sorted lines are written to a log file using **IterFWriteList**. This procedure is *not* recursive. A recursive version of this routine can easily exceed the stack limits when there are lots of nodes in the list.

Since the routine is iterative, the local environment only needs to be loaded once onto the stack. The strategy is to move **TheNode** through the list. At each stopping point, the procedure writes the **Info** field, and is ready to proceed.

Notice that the argument passed as the second parameter to **IterFWriteList** is the linked list, anchored by **Root**. This argument is passed by value, which means that a copy of the address stored in **Root** is passed to the procedure. This is why it's safe to change the value of **TheNode** within **IterFWriteList**. Had this been a **VAR** parameter, you would lose your list by using an iterative loop such as the one in the procedure.

Notice the continuation condition for the **WHILE** loop in the main program. This condition essentially says that the loop should continue as long as there is something to read from the file and as long as there is heap storage available for the material from the file.

Example: Using More Complex Nodes

The complexity of the nodes you can create depends on your needs and your imagination. You can accomplish quite a bit by modifying the basic template for an **LNode**, as described earlier. The **LNode** type you've used in several programs so far is the basis for building *singly linked lists*. This means that each node in the list points to at most one node in the list. In fact, all nodes except the last point to exactly one node.

You can also create **doubly linked lists**, in which each node points to at most two other nodes. A very common example of a doubly linked list is one in which each node points to the next and preceding nodes in the list. The following listing shows the definitions for a node to be used in a doubly linked list and for a pointer to such a node:

```
TYPE DLLPtr = ^DList;
     DList = RECORD
               Info : REAL;
               Next, Preceding : DLLPtr;
             END; (* END DList RECORD *)
```

Lists consisting of such doubly linked nodes are very convenient, since you can traverse the list in both directions, instead of having to start at the beginning of the list each time.

Having two pointers in the record gives you considerable flexibility. For example, you could use the two pointers to build two different lists --- perhaps based on two different items of information. In such a case, each pointer would be used for a different purpose.

For example, in this section you'll find a program that lets you go through a file and count the number of times each word appears in the file. The words are read into a sorted linked list.

A frequency is associated with each word. This represents the number of times the word appears in the file. Thus, the record structure for such a node will have at least the following three fields: a **STRING** field for the word, an **INTEGER** field for frequency, and a pointer field.

In fact, you'll use two pointer fields. One will be used to build a sorted list of words; the second pointer will be used to build a sorted list based on word frequencies. The following declarations will be used in the program:

```
TYPE ShortStr = STRING [ 30];
     DLNodePtr = ^DLNode;
     (* describe a node for a double linked list *)
     DLNode = RECORD
                Wd : ShortStr;
                (* frequency with which Wd appears *)
                Freq : INTEGER;
                StrNext, FreqNext : DLNodePtr;
              END;  (* DLNode RECORD *)
```

As usual, nodes are represented as records. This time the record has four fields instead of two, however. Essentially, you have two records sharing the same structure.

The first ``subrecord" is involved in building the word list, and consists of the **Wd** and **StrNext** fields. The other subrecord is used to build the frequency list, and consists of **Freq** and **FreqNext**.

The **FileStats** program in the next listing does the following:

- Reads from a source file line-by-line.
- Adds individual words from each line to a sorted list of words. A **DLNode** is allocated for each word. If a word is already in the list, the program increases the **Freq** field for

the word's node in the list. If the word is not yet in the list, the program adds the word at the appropriate place in the list.

- After the entire source file has been processed, the program builds a second list, based on word frequencies.

Again, don't be intimidated by the program. Well over half of the code performs auxiliary tasks.

Listing 9.13

```
(* Program to read contents of a file, and to analyze word
   frequencies for the file. Program counts the number of times
   each word found in the file appears. After building a sorted
   list based on the word ordering, the program builds a second
   linked list, based on frequencies.
*)

PROGRAM FileStats;

CONST EmptyString = '';
TYPE CharSet = SET OF CHAR;
     ShortStr = STRING[100];

     DLNodePtr = ^DLNode;
     (* describe a node for a double linked list *)
     DLNode = RECORD
                Wd : ShortStr;
                (* frequency with which Wd appears *)
                Freq : INTEGER;
                StrNext, FreqNext : DLNodePtr;
              END;  (* DLNode RECORD *)

(* Note that there are separate anchors for the
   word and frequency lists.
*)
VAR FreqRoot, Root, Temp : DLNodePtr;
    CurrStr, CurrWd : ShortStr;
    Source, Log : Text;
    SName, LName : STRING;
    NrWds, NrUnique : LONGINT;
    Choice : INTEGER;

(* *****************                    *************** *)
(* ***************** Auxiliary Routines *************** *)
(* *****************                    *************** *)

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  (* Remove the first word from a string;
     a word is separated from the next by 1 or more blanks
  *)
  PROCEDURE RemoveWd ( VAR Wd, Sentence : STRING);
  CONST Blank = ' ';
  VAR Where : INTEGER;
  BEGIN
    (* Remove all leading blanks *)
    WHILE POS ( Blank, Sentence) = 1 DO
      DELETE ( Sentence, 1, 1);
    (* Find the next blank in Sentence.
       This will separate the first two words.
    *)
    Where := POS ( Blank, Sentence);
    IF Where > 0 THEN
    BEGIN
      (* Get the first word, but leave the blank *)
      Wd := COPY ( Sentence, 1, Where - 1);
      (* Remove the first word (and blank) from Sentence *)
      DELETE ( Sentence, 1, Where);
    END
    ELSE  (* If no more blanks --- so only 1 word left *)
    BEGIN
    Wd := Sentence;
    (* Close out the sentence *)
    Sentence := EmptyString;
  END;  (* if Sentence contains no more blanks *)
END;  (* RemoveWd *)

(* Open an existing file, doing all necessary
   and warranted checking, and taking all
   precautions.
   Returns TRUE if file was opened; FALSE otherwise.
*)
FUNCTION FOpened ( VAR TheFile : Text;
                   FName : STRING) : BOOLEAN;
BEGIN
ASSIGN ( TheFile, FName);
(*$I-*)
RESET ( TheFile);
(*$I+*)
IF IOResult = 0 THEN
  FOpened := TRUE
ELSE
  FOpened := FALSE;
END; (* FN FOpened *)

(* Create a new file, doing all necessary
   and warranted checking, and taking all
   precautions.
   Returns TRUE if file was created; FALSE otherwise.
*)
FUNCTION FMade ( VAR TheFile : Text;
                 FName : STRING) : BOOLEAN;
BEGIN
  ASSIGN ( TheFile, FName);
  (*$I-*)
  REWRITE ( TheFile);
  (*$I+*)
  IF IOResult = 0 THEN
    FMade := TRUE
  ELSE
    FMade := FALSE;
```

```
END; (* FN FMade *)

PROCEDURE GetString ( Message : STRING;
                      VAR Value : STRING);
BEGIN
  WRITE ( Message, ' ');
  READLN ( Value);
END; (* GetString *)

(* Write '.' periodically, to let the user know that
   the program is doing something.
   Val is the counter being tested;
   Small is the cycle for a '.'
   Large is the cycle for a WRITELN
*)
PROCEDURE ShowProgress ( Val, Small, Large : LONGINT);
CONST MarkerCh = '.';
BEGIN
  IF ( Val MOD Small = ( Small - 1)) THEN
    WRITE ( MarkerCh);
  IF ( Val MOD Large = ( Large - 1)) THEN
    WRITELN;
END; (* ShowProgress *)

(* Convert characters in TheStr to lower case, if
    necessary.
*)
PROCEDURE MakeStrLower ( VAR TheStr : STRING);
CONST Offset = 32;
VAR Index, HowLong : INTEGER;
BEGIN
  HowLong := LENGTH ( TheStr);
  FOR Index := 1 TO HowLong DO
  BEGIN
    IF TheStr [ Index] IN [ 'A' .. 'Z'] THEN
      TheStr [ Index] := CHR ( ORD (TheStr [ Index]) + Offset);

  END;  (* FOR Index := 1 TO HowLong *)
END;  (* MakeStrLower *)
```

```
(* Replace any characters other than those selected. *)
PROCEDURE CleanStr ( VAR TheStr : STRING;
                     TheSet : CharSet);
CONST Blank = ' ';
VAR Index, HowLong : INTEGER;
BEGIN
  HowLong := LENGTH ( TheStr);
  FOR Index := 1 TO HowLong DO
  BEGIN
    IF NOT ( TheStr [ Index] IN TheSet) THEN
      TheStr [ Index] := Blank;
  END;
END; (* Clean Str *)

(* Get an integer lying within the boundaries
   specified by MinVal and MaxVal
*)
PROCEDURE GetBoundedInt ( Message : STRING;
                          MinVal, MaxVal : INTEGER;
                          VAR Value : INTEGER);

  (* Fix boundary values if they are reversed ---
     that is, if Small > Large.
  *)
  PROCEDURE AdjustVals ( VAR Small, Large : INTEGER);
  VAR Temp : INTEGER;
  BEGIN
    IF Small > Large THEN
    BEGIN
      Temp := Small;
      Small := Large;
      Large := Temp;
    END;
  END; (* AdjustVals *)

  (* Return TRUE if Small <= Val <= Large *)
  FUNCTION IsBetween ( Val, Small, Large : INTEGER) : BOOLEAN;
  BEGIN
    IsBetween := ( Small <= Val) AND ( Large >= Val);
  END;  (* FN IsBetween *)
  BEGIN
    AdjustVals ( MinVal, MaxVal);
    REPEAT
      WRITE ( Message, ' ');
      READLN ( Value);
    UNTIL IsBetween ( Value, MinVal, MaxVal);
  END; (* GetBoundedInt *)

(* *****************                      *************** *)
(* ***************** Linked List Routines *************** *)
(* *****************                      *************** *)

  (* get space for and initialize a doubly linked list node *)
  PROCEDURE InitDLNode ( VAR TheNode : DLNodePtr);
  BEGIN
    NEW ( TheNode);
    TheNode^.Wd := EmptyString;
    TheNode^.Freq := 0;
    TheNode^.FreqNext := NIL;
    TheNode^.StrNext := NIL;
  END; (* InitDLNode *)

  PROCEDURE IterFWriteDList ( VAR TheFile : Text;
                              TheNode : DLNodePtr);
  VAR Count : INTEGER;
  BEGIN
    Count := 0;
    WHILE TheNode <> NIL DO
    BEGIN
      WRITELN ( TheFile, TheNode^.Wd, ' : ', TheNode^.Freq);
      INC ( Count);
      TheNode := TheNode^.StrNext;
    END;
    WRITELN ( Count : 4, ' entries in word list');
  END; (* IterFWriteDList *)

  PROCEDURE iIterFWriteDList ( VAR TheFile : Text;
                               TheNode : DLNodePtr);
  VAR Count : INTEGER;
```

```
BEGIN
  Count := 0;
  WHILE TheNode <> NIL DO
  BEGIN
    WRITELN ( TheFile, TheNode^.Freq : 4, ' : ', TheNode^.Wd);
    INC ( Count);
    TheNode := TheNode^.FreqNext;
  END;
  WRITELN ( Count : 4, ' entries in frequency list');
END; (* iIterFWriteDList *)

(* insert new node (ToAdd) in lexicographic position in a
   linked list (WhereToAdd)
*)
PROCEDURE DLLInMiddle ( ToAdd : DLNodePtr;
                        VAR WhereToAdd : DLNodePtr);
BEGIN
  (* if at the end, (and not yet added,) add the node *)
  IF ( WhereToAdd = NIL) THEN
    WhereToAdd := ToAdd
  (* else if the node should be added at the current spot *)
  ELSE IF ( WhereToAdd^.Wd > ToAdd^.Wd) THEN
  BEGIN
    ToAdd^.StrNext := WhereToAdd;
    WhereToAdd := ToAdd;
  END
  (* else if the word is already in the list,
     just increase the frequency field.
  *)
  ELSE IF ( WhereToAdd^.Wd = ToAdd^.Wd) THEN
  BEGIN
    INC ( WhereToAdd^.Freq);
  END
  ELSE (* if still not able to add node *)
    DLLInMiddle ( ToAdd, WhereToAdd^.StrNext);
END; (* DLLInMiddle *)

(* insert new node (ToAdd) in position in a
   linked list (WhereToAdd)
*)
```

```
  PROCEDURE iDLLInMiddle ( ToAdd : DLNodePtr;
                           VAR WhereToAdd : DLNodePtr);
  BEGIN
    (* if at the end, (and not yet added,) add the node *)
    IF ( WhereToAdd = NIL) THEN
      WhereToAdd := ToAdd
    (* else if the node should be added at the current spot *)
    ELSE IF ( WhereToAdd^.Freq <= ToAdd^.Freq) THEN
    BEGIN
      ToAdd^.FreqNext := WhereToAdd;
      WhereToAdd := ToAdd;
    END
    ELSE (* if still not able to add node *)
      iDLLInMiddle ( ToAdd, WhereToAdd^.FreqNext);
  END; (* iDLLInMiddle *)

BEGIN  (* Main program *)
  NEW ( Root);
  DISPOSE ( Root);
  Root := NIL;
  FreqRoot := NIL;
  Temp := NIL;
  NrWds := 0;
  NrUnique := 0;
  GetString ( 'Source file?', SName);
  GetString ( 'Log file?', LName);
  WRITELN ( 'After initializing the node pointers, ');
  WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');

  IF FOpened ( Source, SName) AND FMade ( Log, LName) THEN
  BEGIN
    WHILE ( NOT EOF ( Source)) AND
          ( MEMAVAIL > SIZEOF ( DLNode)) DO
    BEGIN
      (* Read the source file, line by line *)
      READLN ( Source, CurrStr);
      (* homogenize input by making everything lowercase *)
      MakeStrLower ( CurrStr);
      (* Replace any unusual characters with blanks;
         you may want to remove this line.
    *)
    CleanStr ( CurrStr, [ 'a' .. 'z', 'A' .. 'Z', '-', '''']);
    (* process each word on the line; add the word to the
       list, if warranted.
    *)
    REPEAT
      RemoveWd ( CurrWd, CurrStr);
      IF CurrWd <> EmptyString THEN
      BEGIN
        INC ( NrWds);
        ShowProgress ( NrWds, 5, 250);
        (* Allocate storage for the word *)
        InitDLNode ( Temp);
        Temp^.Wd := CurrWd;
        (* The word has appeared at least once *)
        Temp^.Freq := 1;
        (* Add the word to the list *)
        DLLInMiddle ( Temp, Root);
      END;
    UNTIL CurrStr = EmptyString;
  END;  (* WHILE NOT EOF ( Source) AND still heap available *)
  IF NOT EOF ( Source) THEN
    WRITELN ( 'Terminated early; not enough memory.');
  WRITELN;
  WRITELN ( NrWds : 10, ' words');

  (* Start at the front of the WORD list *)
  Temp := Root;
  (* While not every word has been processed *)
  WHILE Temp <> NIL DO
  BEGIN
    (* Build a sorted list using Frequency fields.
       NOTE use of FreqRoot, since this will be a
       second list --- distinct from the word list.
       Notice, however, that no storage is allocated
       for this second list. The frequency list shares
       the word list's storage.
    *)
    iDLLInMiddle ( Temp, FreqRoot);
    (* The word list will be used to provide the nodes,
         so Temp must follow the StrNext pointers.
      *)
      Temp := Temp ^.StrNext;
    END;

    (* Find out what information user wants to save. *)
    GetBoundedInt ( 'Save by 1) word;  2) frequency; 3) both',
                    1, 3, Choice);
    CASE Choice OF
      1 :
        IterFWriteDList ( Log, Root);
      2 :
        iIterFWriteDList ( Log, FreqRoot);
      3 :
        BEGIN
          IterFWriteDList ( Log, Root);
          WRITELN ( Log, '-----------------------------------');
          iIterFWriteDList ( Log, FreqRoot);
        END; (* 3 *)
    END; (* CASE OF Choice *)
    WRITELN ( 'After reading from the file, ');
    WRITELN ( 'MEMAVAIL = ', MEMAVAIL : 10, ' bytes');
    CLOSE ( Source);
    CLOSE ( Log);
  END (* IF FOpened and FMade *)
  ELSE
    WRITELN ( 'File error.');
  AwaitUser;
END.
```

Let's work through the list building process for each list. The word list is built first. Storage for the nodes is allocated during this part. For each word the program finds while reading **Source**, InitDLNode is called.

After the necessary assignments and updating have been done, the program adds the new word to the word list. This process involves **DLLInMiddle**. This procedure is similar to versions in other programs, but contains an extra **ELSE IF** clause. This clause handles the case where the new word is the same as one in the word list --- that is, where

```
WhereToAdd^.Wd = ToAdd^.Wd
```

When this happens, the node is not added to the list. Instead, the value of the **Freq** field of **WhereToAdd^** is incremented by 1. Note that even though the node is not added to the list, the storage for the node is not reused in the example program. This storage will be lost when **InitDLNode** is called again. (You might want to try modifying the program so that you can give back unused storage. This will enable you to process a greater number of different words.)

The frequency list is anchored by **FreqRoot**. This list is built as follows:

- The program traverses the *word* list, to make sure every node is added to the frequency list. **Temp** is used to loop through the word list.
- For each node in the word list, the program adds the node to the frequency list, based on the value of the node's **Freq** field. Procedure **iDLLInMiddle** builds these lists.

Notice that **FreqRoot** is the second parameter to **iDLLInMiddle**, not **Root**. This is necessary, since the word and frequency lists must have different anchors. Even though both lists contain the same nodes, these won't be in the same order.

Notice also that there are no calls to **InitDLNode** in the second part of the **FileStats** program. This is because all the required nodes have already been allocated. The second part of the program takes advantage of the fact that a **DLNode** has two different pointer fields. This enables you to get ``double mileage" out of the nodes.

The program uses **IterFWriteList** and **iIterFWriteList** to write the word and frequency lists, respectively. Both of these procedures are straightforward.

Summary

In this chapter, you learned about Pascal's pointer types and about how to use them. The major advantage of pointers is that they enable you to use the heap for your program's memory. You can allocate and deallocate storage to suit the needs of your program.

Pointers also enable you to define self-referential types, which can be used as components in building structures of indeterminate size. In the chapter, you learned how to build linked lists using this technique.

If you're interested, you might want to explore other data structures involving pointers --- for example, trees. To learn about such data types, look at a more advanced book, such as textbooks used for data structures courses in Computer Science.

In the next chapter you'll learn about CodeWarrior Pascal's predefined units.

10

Units and Bits, Oh My

In the first part of this chapter, you'll learn about units. These are the files you've been using --- pretty much without explanation --- in many of the example programs. In the rest of this chapter you'll learn about some miscellaneous topics that have not been covered elsewhere.

As you've seen, CodeWarrior Pascal has dozens of predefined procedures and functions that you can call in your programs. These include routines for string-handling (**INSERT**, **COPY**, etc.), for mathematical operations (**ARCTAN**, **COS**, etc.), and for miscellaneous tasks (**READ**, **WRITELN**, etc.). A large collection of these routines is included in the CodeWarrior Pascal runtime library (RTL), and you don't need to take any special actions to make these routines accessible to your programs.

However, sometimes you need to do things that are a bit more specialized, and for which the RTL lacks routines. In such cases, you can make use of *units* in the Pascal environment. Units are special-purpose packages that contain the definitions and routines you need to carry out a specific task. Once you've created such a unit, you can include it in other programs. In this chapter, you'll learn how to create your own units.

Before that, however, we'll take a brief look at what a unit looks like, and you'll learn a bit about the units included in your CodeWarrior Pascal environment.

Units : Public Interface and Private Implementation

There are two parts to every unit: an interface and an implementation. The *interface*, as its name suggests, allows your program to get access to something --- in this case, the code contained in the unit's implementation. In fact, the interface file consists mainly of interfaces, and possibly also a few definitions.

The Interface Shows what a Unit Has to Offer ...

The greatly overused term interface has several different definitions, all of which have to do with some sort of boundary between two elements:

- It refers to the component through which you interact with your computer. For example, the Windows Graphical User Interface, or GUI, is what you use to interact with the files and applications on your computer.
- It refers to a type of file through which your program gains access to routines and resources stored elsewhere. For example, **Windows.P** and **Functions.P** are interface files that give your program access to various functions in addition to those in the runtime library.
- It refers to the mechanism through which routines in your program pass information to each other. In Chapter 4, you'll learn about procedures and functions, and also about the use of parameters to pass information between routines. You can think of parameters as slots between two sides of a wall or between two rooms. The size and shape (so to speak) of these slots will determine the kind of information you can pass through them and also the direction in which information can travel.

Interestingly, one reason for the use of interface files (second item) is to let the compiler check that the format for procedure and function calls in your program match the interfaces (item three) for the called routines. That is, the compiler uses the information in an interface file to make sure you are calling routines correctly.

The following listing contains the first few lines from the **Functions.P** interface file included with CodeWarrior Pascal. Material between (* and *) represents a comment, and is ignored by any program that processes the interface file.

```
(*
 * Metrowerks Pascal Interfaces For Windows.
 * Metrowerks inc, c1996.
 *)

UNIT Functions;

INTERFACE

USES
  Types;

(* * * * * * * * * * * * * External C functions * * *  * * * * *
* * * * * *)

PROCEDURE abort; C; EXTERNAL;
FUNCTION abs(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION atof(a0:UNIV LONGINT):DOUBLE; C; EXTERNAL;
FUNCTION atoi(a0:UNIV LONGINT):LONGINT; C; EXTERNAL;
FUNCTION atol(a0:UNIV LONGINT):LONGINT; C; EXTERNAL;
FUNCTION calloc(a0:size_t; a1:size_t):PVOID; C; EXTERNAL;
PROCEDURE free(a0:UNIV LONGINT); C; EXTERNAL;
FUNCTION isalnum(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION isalpha(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION iscntrl(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION isdigit(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION isgraph(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION isleadbyte(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION islower(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION isprint(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION ispunct(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION isspace(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION isupper(a0:LONGINT):LONGINT; C; EXTERNAL;
FUNCTION iswalnum(a0:wchar_t):LONGINT; C; EXTERNAL;
FUNCTION iswalpha(a0:wchar_t):LONGINT; C; EXTERNAL;
FUNCTION iswascii(a0:wchar_t):LONGINT; C; EXTERNAL;
FUNCTION iswcntrl(a0:wchar_t):LONGINT; C; EXTERNAL;

(* ** * * * * * * * Windows functions * * * * * * * * * * * * *
* * * * *)

{A}
FUNCTION AbortPath(a0:HDC):BOOL; STDCALL; DLLIMPORT;
FUNCTION AbortPrinter(a0:HANDLE):BOOL; STDCALL; DLLIMPORT;
FUNCTION AbortSystemShutdownA(a0:LPSTR):BOOL; STDCALL; DLLIMPORT;
FUNCTION AbortSystemShutdownW(a0:LPWSTR):BOOL; STDCALL;
DLLIMPORT;
FUNCTION accept(a0:_SOCKET; VAR a1:sockaddr; VAR
a2:LONGINT):_SOCKET;
FUNCTION AcceptEx(a0:_SOCKET; a1:_SOCKET; a2:PVOID; a3:DWORD;
a4:DWORD; a5:DWORD; a6:LPDWORD; a7:LPOVERLAPPED):BOOL;
FUNCTION AccessCheckAndAuditAlarmA(a0:LPCSTR; a1:LPVOID;
a2:LPSTR; a3:LPSTR; a4:PSECURITY_DESCRIPTOR; a5:DWORD;
a6:PGENERIC_MAPPING; a7:BOOL; a8:LPDWORD; a9:LPBOOL;
b0:LPBOOL):BOOL; STDCALL; DLLIMPORT;
FUNCTION AccessCheckAndAuditAlarmW(a0:LPCWSTR; a1:LPVOID;
a2:LPWSTR; a3:LPWSTR; a4:PSECURITY_DESCRIPTOR; a5:DWORD;
a6:PGENERIC_MAPPING; a7:BOOL; a8:LPDWORD; a9:LPBOOL;
b0:LPBOOL):BOOL; STDCALL; DLLIMPORT;
FUNCTION AddAtomA(a0:LPCSTR):ATOM; STDCALL; DLLIMPORT;
FUNCTION AddAtomW(a0:LPCWSTR):ATOM; STDCALL; DLLIMPORT;
FUNCTION AddFormA(a0:HANDLE; a1:DWORD; a2:LPBYTE):BOOL; STDCALL;
DLLIMPORT;
FUNCTION AddFormW(a0:HANDLE; a1:DWORD; a2:LPBYTE):BOOL; STDCALL;
DLLIMPORT;
FUNCTION AddMonitorA(a0:LPSTR; a1:DWORD; a2:LPBYTE):BOOL;
STDCALL; DLLIMPORT;
FUNCTION AddMonitorW(a0:LPWSTR; a1:DWORD; a2:LPBYTE):BOOL;
STDCALL; DLLIMPORT;
FUNCTION AddPortA(a0:LPSTR; a1:HWND; a2:LPSTR):BOOL; STDCALL;
DLLIMPORT;
FUNCTION AddPortW(a0:LPWSTR; a1:HWND; a2:LPWSTR):BOOL; STDCALL;
DLLIMPORT;
FUNCTION AddPrinterA(a0:LPSTR; a1:DWORD; a2:LPBYTE):HANDLE;
STDCALL; DLLIMPORT;
FUNCTION AddPrinterW(a0:LPWSTR; a1:DWORD; a2:LPBYTE):HANDLE;
STDCALL; DLLIMPORT;
```

Note that units can make use of other units. This is where the *Uses Propagation* compiler option becomes important. By turning this option on, you tell the compiler to look at **Types.P** (if it hasn't done so already), because **Functions.P** uses some of the routines or data types defined there.

The rest of the listing consists of the interfaces for various procedures and functions that this unit makes available. As it turns out, **Functions.P** gives you access to over 1000 C and Windows routines. The routines in the listing are all available in C libraries that are included in your CodeWarrior environment.

The contents of an interface file tell you what functions and data types you can use by including the unit. You can think of the interface file as similar to a super-informative directory listing --- for example, such you might find in the lobby of an office building. However, an interface file is more complete than such a directory. To be as informative as an interface file, the directory would have to do more than just listing the names of all the people available to the public. For example, the directory might also include vital information such as office hours and phone numbers --- so anyone interested would know how to contact the person.

... The Implementation Delivers on the Promise

If the interface is the building directory, then the implementation is the offices listed in the directory --- and possible some additional ones as well. These "unlisted" offices might provide services to the building's tenants but not to the general public. Moreover, an office that has one listing in the directory may have several offices related to that listing.

To return from the office analogy, the implementation file has the routines that are advertised in the interface. All the details are contained in the implementation. Implementation routines may use procedures, functions, and data types that are in the unit but that are not listed in the interface. In Pascal, routines may also have (sub)routines defined within them. For example, a function to convert between Celsius and Fahrenheit temperature readings might actually include two subroutines --- **CelsToFahr** and **FahrToCels** --- to make the two conversions. The function would call the appropriate one for the desired conversion. The user never even has to know that these two routines exist.

The CodeWarrior Pascal Units

The CodeWarrior environment includes more than just Pascal resources. For example, it also includes a C/C++ programming environment. By using units, the folks who created CodeWarrior Pascal have managed to provide you with access to many of these additional resources.

For example, they've provided you with more data types than standard Pascal, and they've also managed to give your programs access to over 1000 procedures and functions --- over and above the routines available in the Pascal runtime library (RTL).

WARNING! Make sure you don't change any of the unit files if you examine them with a text editor. In fact, to be on the safe side, it's probably a good idea to copy any such files into a temporary directory and then to examine the copies.

The CodeWarrior Pascal environment includes the following interface files:

- **Windows.P**, which is just a file to tell the compiler about all the other interface files. If you look at the file (located in the **Interface** directory of your CodeWarrior environment), you'll see that the Windows Unit interface consists of a single **USES** statement that includes all the other interface files.
- **Types.P**, which defines additional data types. These extra types include several types that are defined in the C and C++ languages --- for example, **INT** (for **INTEGER**) and **FLOAT** (for **REAL**).
- **Constants.P**, **Variables.P**, and **Macros.P**, and contain definitions useful for various

types of systems and other low-level programming tasks. You probably won't ever need to use directly any of the macros defined in these files. However, routines that your programs call may use them.

- **Functions.P**, which gives you access to a few dozen C routines, as well as to over 1000 routines for programming in the Windows environment. The **SRAND** and **RAND** routines used in several of the example programs are both part of the C environment, and are accessed through the Functions unit.

Including the CodeWarrior Pascal Units

To get access to the definitions provided in the units, you need to include a **USES** statement immediately after your program heading. The most convenient way to do this is to specify **Windows** as the unit used. Since this file uses all the other units, the compiler will automatically know to search for all the unit files.

You still need to include these files in your project. To do this, select the Add Files command from the Project menu. Then, select the unit files and add them to the project.

For example, the following program uses the **FUNCTIONS** unit (**Functions.P**). For this project, you should add all the interface files except **Windows.P**.

Listing 10.1

```
PROGRAM UnitEg1;
USES Functions;

CONST EmptyString = '';
VAR CurrStr : STRING;
    SLen, NrAlpha, NrDigits, NrLower, NrUpper, NrOther : INTEGER;

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to end program');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE ZeroVals;
  BEGIN
    SLen := 0;
    NrAlpha := 0;
    NrDigits := 0;
    NrLower:= 0;
    NrUpper := 0;
  END; (* ZeroVals *)

  PROCEDURE StudyStr (TheStr : STRING);
  VAR Index, SLen : INTEGER;
  BEGIN
    ZeroVals;
    SLen := LENGTH ( TheStr);
    FOR Index := 1 TO SLen DO
    BEGIN
      IF IsAlpha ( ORD ( TheStr [ Index])) <> 0 THEN
      BEGIN
        INC ( NrAlpha);
        IF IsUpper ( ORD ( TheStr [ Index])) <> 0 THEN
          INC ( NrUpper)
        ELSE
          INC ( NrLower);
      END  (* if character is a letter *)
      (* ELSE if character is a digit *)
      ELSE IF IsDigit ( ORD ( TheStr [ Index])) <> 0 THEN
        INC ( NrDigits)
      ELSE  (* if character is neither letter nor digit *)
        INC ( NrOther);
    END;
  END;  (* StudyStr *)

  PROCEDURE ReportResults;
  BEGIN
    WRITELN ( NrAlpha : 2, ' letters (',
              NrUpper : 2, ' uppercase, ', NrLower : 2, '
lowercase); ',
              NrDigits : 2, ' digits; ', NrOther : 2, ' other');

  END;   (* ReportResults *)
```

```
BEGIN
  WRITELN ( 'A string to study, please: (ENTER ends the loop) ');
  READLN ( CurrStr);
  WHILE CurrStr <> EmptyString DO
  BEGIN
    StudyStr ( CurrStr);
    ReportResults;
    WRITELN ( 'A string to study, please: (ENTER ends the loop)
');
    READLN ( CurrStr);
  END;
  AwaitUser;
END.
```

Creating Your Own Units

You can create units of your own --- to hold special-purpose or commonly-used routines. In this section, you'll learn how to do this.

Example: A Small Utilities Unit

Recall that a unit consists of two main parts: an interface and an implementation. To illustrate these parts, let's look at a small example. The following listing shows a utilities unit which contains just two functions, and which defines two constants.

Listing 10.2

```
(* Unit DemoUnit: Pascal utilities. *)

UNIT DemoUnit;
```

```
(* Specify type definitions and variable declarations;
   list procedure and function headings (interfaces).
*)
INTERFACE

CONST InvalidVal = -99999.99;
      Tolerance =  1.0E-6;

(* Interfaces for routines available through the unit *)
FUNCTION NonZero ( Val : REAL) : BOOLEAN;
FUNCTION SafeDivision ( Numer, Denom : REAL) : REAL;

(* Define the routines listed in interface, as well as
   any others that are private to the unit.
*)
IMPLEMENTATION

(* Return TRUE if Val differs from zero by more
   than a predefined amount.
*)
FUNCTION NonZero ( Val : REAL) : BOOLEAN;
BEGIN
  IF ABS ( Val - 0.0) > Tolerance THEN
    NonZero := TRUE
  ELSE
    NonZero := FALSE;
END; (* FN NonZero *)


(* Divide num by denom, checking for division by zero
   before doing so. Return quotient or a default value
   ( InvalidVal) on division by zero.
*)
FUNCTION SafeDivision ( Numer, Denom : REAL) : REAL;
BEGIN
  IF NonZero ( Denom) THEN
    SafeDivision := Numer / Denom
  ELSE
    SafeDivision := InvalidVal;
```

```
END; (* SafeDivision *)

(* A routine available only within DemoUnit,
   NOT to any other program.
*)
PROCEDURE APrivateProc;
BEGIN
  WRITELN ( 'Hello from Util unit.');
END; (* APrivateProc *)

END.   (* Note the END, followed by a period *)
```

This example contains an interface and an implementation. The unit's name and the name of the file in which the unit is stored must be the same. Thus, **DemoUnit** must be stored in a file named DEMOUNIT. To distinguish units from program files, we use the extension **P** for units. Thus, unit **DemoUnit** is stored in file **DEMOUNIT.P**. This unit is not included as a project, but you can easily create a project and a compiled unit.

The first part of the unit source file will contain the interface. In **DemoUnit**, two constants are defined, and two functions are declared. Because these objects are specified in the interface, they are available to any programs that use the unit --- just as if the constants had been defined in the main program and the routine headers had been included as a **FORWARD** declaration --- so that the definition would come later. (The word **FORWARD** is not used in the unit's interface, however.)

The remainder of the file contains the implementation for the unit --- that is, the procedure and function definitions.

There are several points to notice about the implementation:

- Both the interface and the implementation include parameter lists for the functions. They are required in the interface --- because the compiler uses the information in the interface to evaluate the correctness of your calls to the unit's routines. The parameter lists are optional in the implementation. If you use them, however, the lists in the interface and the implementation must match exactly; otherwise the compiler will respond with an error.
- The implementation contains a third routine --- **APrivateProc**. This procedure is not available to outside programs, since it was not declared in the interface. Such a routine is said to be *private* or *hidden*. This procedure is included here just to make this point. If a unit does contain private routines, these could be called only by routines defined in the unit's implementation.

- The unit file ends with **END** followed by a period.

Once you've written the code for the unit, you need to compile. The general procedure for doing this is the same as for any other program --- except that certain of the project settings need to be changed. In particular, the unit must be linked differently.

In the x86 Linker section of the Project Settings dialog box, you need to change the value for *Entry Point Usage*. For programs, this value is set to Default, but for units it must be set to None.

TIP: When compiling a unit, set the *Entry Point Usage* option to None in the x86 Linker part of the Project Settings dialog box.

Once the unit file has been created, you can use the unit's routines in your other programs. The following listing illustrates this.

Listing 10.3

```
PROGRAM DUTest;
USES DemoUnit;

VAR V1, V2, Result : REAL;

  PROCEDURE GetReal ( Message : STRING;
                      VAR Value : REAL);
  BEGIN
    WRITE ( Message, '? ');
    READLN ( Value);
  END; (* GetReal *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to end program.');
    READLN (Ch);
  END;  (* AwaitUser *)

BEGIN
  GetReal ( 'Value 1', V1);
  GetReal ( 'Value 2', V2);
  IF NonZero ( V2) THEN
    WRITELN ( 'V2 is non-zero')
  ELSE
    WRITELN ( 'V2 is too close to zero.');

  WRITELN ( 'V1 / V2 = ', SafeDivision ( V1, V2) : 10 : 5);
  AwaitUser;
END.
```

To compile, link, and run this project, you first need to compile the **DEMOUNIT.P** file. Remember to set Entry Point to None in the x86Linker Tab in the Project Settings, and remember to name the project **DEMOUNIT.CWP**.

As you develop a collection of procedures and functions, you'll probably want to turn some of them into units --- for convenience and ease of use.

Bit Handling

As you know, numerical values are stored as sequences of bits in CodeWarrior Pascal. For whole number types (**BYTE**, **INTEGER**, **LONGINT**, and so forth), CodeWarrior Pascal provides operators that lets you work with individual bits of such representations. These operators take one or two whole number operands and produce a result of the same type. The operators themselves are denoted using the same words as for the logical operators.

Bitwise Negation: The NOT Operator

The arithmetic negation operator reverses the sign of a number. For example, 35 becomes -35; -59 becomes +59. CodeWarrior Pascal's *bitwise negation*, or **NOT**, operator reverses the value of each bit in a number's representation. This operator is denoted using the reserved word **NOT**.

The bitwise **NOT** is a unary operator which takes a whole number operand and returns another number. If a particular bit in the original number was 1, this bit is 0 in the resulting number; conversely, if the original bit was 0, the corresponding bit in the new value is 1.

Although this operator has the same notation as the logical negation operator, the two operators are quite different, and their operands cannot be substituted for each other. The logical negation operator changes a true to a false and vice versa; the bitwise negation operator changes 0 to 1 and vice versa, throughout the representation of a number.

The following program illustrates the bitwise **NOT** operator, and compares it to the arithmetic negation operator. Don't worry about the **IntBits** function for now. It's included in these early programs because it provides a clean way of determining the bit pattern for a whole number.

Listing 10.4

```
(* Program to illustrate use of bitwise NOT operator *)

PROGRAM BitNotDemo;
USES Types;

VAR Val : INTEGER;

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* Await User *)

  (* Determine and return bit pattern for specified value. *)
  FUNCTION IntBits ( Value : WORD) : STRING;
  TYPE PattArray = ARRAY [ 0 .. 15] OF STRING [ 4];
  CONST (* Used to get individual hexadecimal digits *)
        Mask1 = $F;          Mask2 = $F0;
        Mask3 = $F00;        Mask4 = $F000;
```

```
VAR  Patterns : PattArray;
     Temp, Piece : STRING;

   PROCEDURE InitPatterns;
   BEGIN
     Patterns [ 0] := '0000';
     Patterns [ 1] := '0001';
     Patterns [ 2] := '0010';
     Patterns [ 3] := '0011';
     Patterns [ 4] := '0100';
     Patterns [ 5] := '0101';
     Patterns [ 6] := '0110';
     Patterns [ 7] := '0111';
     Patterns [ 8] := '1000';
     Patterns [ 9] := '1001';
     Patterns [ 10] := '1010';
     Patterns [ 11] := '1011';
     Patterns [ 12] := '1100';
     Patterns [ 13] := '1101';
     Patterns [ 14] := '1110';
     Patterns [ 15] := '1111';
   END; (* InitPatterns *)

BEGIN  (* IntBits *)
  InitPatterns;
  Temp := '';

  Piece := Patterns [ Value AND Mask4 SHR 12];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask3 SHR 8];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask2 SHR 4];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask1];
  Temp := Temp + Piece;

  IntBits := Temp;
END; (* FN IntBits *)
```

```
BEGIN
  REPEAT
    GetInt ( 'Value? (0 to stop)', Val);
    WRITELN ( Val : 10, IntBits ( Val) : 20);
    WRITELN ( 'NOT ', Val : 6, IntBits ( NOT Val) : 20,
              '(' : 4, NOT Val : 6, ')');
    WRITELN ( -Val : 10, IntBits ( -Val) : 20);
  UNTIL Val = 0;
  AwaitUser;
END.
```

Run the program with the following kinds of values:

- A small and a large positive number.
- A small and a large negative number.
- Two consecutive positive values.
- The same two consecutive negative values.
- **MaxInt** and **MaxInt** + 1.
- Any other values you're curious about.

The program displays bit patterns for the original value, its bitwise negation and its arithmetic negation. Next to the bitwise negation, you'll also find the numerical value corresponding to the bit pattern.

Notice the relationship between the bit patterns and the values of the bitwise and arithmetic negation. The arithmetic negation is simply the bitwise negation + 1. If you're curious as to why this is so, you should learn about *one's* and *two's complement* representations for numbers. (Books on discrete mathematics and computer science textbooks will have information about these representations.)

The bitwise **NOT** operator has the same high order of precedence as the logical **NOT** and other unary operators.

Bitwise Multiplication: The Bitwise AND Operator

There are several binary bitwise operators. These combine corresponding bits of two whole number operands to produce a bit for the resulting number. For example, the bitwise **AND** operator multiplies corresponding bits from its two operands, and returns the product (0 or 1) as a bit value for the number being built.

The following program shows how to use the bitwise **AND** operator. Run the program for different combinations of values. In particular, try two positive, two negative, and two numbers of opposite sign. What is the sign of the result in each instance?

Listing 10.5

```
(* Program to illustrate use of bitwise AND operator *)

PROGRAM BitAndDemo;
USES Types;

VAR Val1, Val2, Result : INTEGER;

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END;  (* AwaitUser *)

  (* Determine and return bit pattern for specified value. *)
  FUNCTION IntBits ( Value : WORD) : STRING;
  TYPE PattArray = ARRAY [ 0 .. 15] OF STRING [ 4];
  CONST (* Used to get individual hexadecimal digits *)
        Mask1 = $F;          Mask2 = $F0;
        Mask3 = $F00;        Mask4 = $F000;
```

```
VAR  Patterns : PattArray;
     Temp, Piece : STRING;

   PROCEDURE InitPatterns;
   BEGIN
     Patterns [ 0] := '0000';
     Patterns [ 1] := '0001';
     Patterns [ 2] := '0010';
     Patterns [ 3] := '0011';
     Patterns [ 4] := '0100';
     Patterns [ 5] := '0101';
     Patterns [ 6] := '0110';
     Patterns [ 7] := '0111';
     Patterns [ 8] := '1000';
     Patterns [ 9] := '1001';
     Patterns [ 10] := '1010';
     Patterns [ 11] := '1011';
     Patterns [ 12] := '1100';
     Patterns [ 13] := '1101';
     Patterns [ 14] := '1110';
     Patterns [ 15] := '1111';
   END; (* InitPatterns *)

BEGIN  (* IntBits *)
  InitPatterns;
  Temp := '';

  Piece := Patterns [ Value AND Mask4 SHR 12];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask3 SHR 8];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask2 SHR 4];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask1];
  Temp := Temp + Piece;

  IntBits := Temp;
END; (* FN IntBits *)

(* Draw a line of a specified length on the screen.
     The line will be built of TheCh.
  *)
  PROCEDURE DrawLine ( Size : INTEGER;
                       TheCh : CHAR);
  VAR Index : INTEGER;
  BEGIN
    FOR Index := 1 TO Size DO
      WRITE ( TheCh);
    WRITELN;
  END; (* DrawLine *)

BEGIN
  REPEAT
    GetInt ( 'Value 1? (0 to stop)', Val1);
    IF Val1 <> 0 THEN
    BEGIN
      GetInt ( 'Value 2?', Val2);
      Result := Val1 AND Val2;
      WRITELN ( ' ':6, IntBits (Val1) :16, ' (', Val1 : 6, ')');
      WRITELN ( 'AND   ', IntBits ( Val2) : 16, ' (',
                Val2 : 6, ')');
      DrawLine ( 25, '=');
      WRITELN ( ' ':6, IntBits (Result) :16, ' (',
                Result : 6, ')');
    END;
  UNTIL Val1 = 0;
  AwaitUser;
END.
```

Notice that the bitwise **AND** operator yields a 1 only if the bits from both operands are 1. That is, 1 * 1 is the only product of 0 and 1 that equals 1.

Masks

The bitwise **AND** operator serves to exclude bits from the result. Not surprisingly, a common use for this operator is to do things such as converting a 16-bit value to an 8-bit one or, more generally, removing particular portions of a number's bit pattern.

The process of removing certain bits in a number is called *masking*. The easiest way to mask is to apply the bitwise **AND** operator, using an appropriate bit pattern as the *mask*. The mask is a bit pattern that has a 0 at every position you want to remove from your number and a 1 everywhere else.

For example, to get an 8-bit value (between 0 and 255) from an integer, you could use the following mask (but without the spaces, which are included for clarity).

```
0000 0000 1111 1111
```

By applying the bitwise **AND** operator to your original value and this mask, you'll get a result that has 1s only in the rightmost byte. No matter what bit pattern your number has in the left byte, the mask will ensure that the resulting bits are all 0.

Note that the mask value is 255, or $FF, in hexadecimal. Note also that you're actually getting back an integer --- that is, a 16-bit value. It just happens to be an integer that has all 0s in its left byte.

Function **Mask** in the following listing lets you carry out such modifications. The parameters and return type are of type **WORD** instead of **INTEGER** so that it's possible to use masks that involve the leftmost bit without having to worry about signs.

```
FUNCTION Mask ( Value, MaskPattern : WORD) : WORD;
BEGIN
  Mask := Value AND MaskPattern;
END; (* FN Mask *)
```

Specifying a Mask

The mask you'll want to use in a particular situation will have a specific bit pattern; however, the **MaskPattern** parameter to the function is an **INTEGER**. You may find it easier to specify your mask value in hexadecimal form.

To determine the hexadecimal form of the mask you want, proceed as follows:

- Determine the binary representation your mask should have.
- Group the digits from this representation into groups of four bits.

- For each group of bits, determine the hexadecimal digit that corresponds to the bit pattern. You can use the **Pattern** array defined in **IntBits** to determine the digit that corresponds to a particular bit pattern.

The resulting value for each group of bits will be a hexadecimal digit --- 0 .. 9 or A .. F.

Each **INTEGER** mask will have up to four hexadecimal digits. (A mask will have fewer than four if any leading hexadecimal digits are 0.) A mask for a **LONGINT** would have eight hexadecimal digits, since 32 bits are used to represent such a value.

For example, suppose you wanted to generate a random value between 0 and 64. One way to do this would be to generate a random integer and then to mask out everything but six consecutive bits somewhere within the number. The following bit pattern would get digits 5 through 10.

```
0000 0011 1111 0000
```

The hexadecimal digits for this bit pattern are: 0, 3, F (15), and 0. Thus, the mask would be $3F0. This is much easier to determine than 1008, the decimal value corresponding to this bit pattern.

The following program lets you generate random integers and mask these values in two different ways. Run the program and see whether you notice any pattern in the results.

Listing 10.6

```
(* Program to illustrate use of masks *)

PROGRAM RandomAndDemo;
USES Windows;

CONST OnOff = $AAAA;
      OffOn = $5555;
VAR Index, NrTrials, Value, RandSeed : INTEGER;

  (* Return the result when all but the bits in MaskPattern
     are removed from Value.
  *)
  FUNCTION Mask ( Value, MaskPattern : WORD) : WORD;
  BEGIN
    Mask := Value AND MaskPattern;
  END; (* FN Mask *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN
  GetInteger ( ’Integer seed for random number generator’,
               RandSeed);
  SRAND (RandSeed);
  GetInteger ( '# of Trials?', NrTrials);
  FOR Index := 1 TO NrTrials DO
  BEGIN
    Value := RAND;
    WRITELN ( Value : 5, ': ', Mask ( Value, OnOff) : 5, ' + ',
              Mask ( Value, OffOn) : 5);
  END;
  READLN;
END.
```

This program produced the following output for a sample run:

```
# of Trials? 15
 9676:  8328 +  1348
25162:  8714 + 16448
18097:   672 + 17425
12514:  8354 +  4160
18849:  2208 + 16641
26203:  8714 + 17489
28461: 10792 + 17669
24289:  2720 + 21569
 6960:  2592 +  4368
```

```
30914: 10370 + 20544
 5113:   680 +  4433
30076:  8232 + 21844
 8987:  8714 +   273
18326:   642 + 17684
27186: 10786 + 16400
```

Notice that the mask results add up to the original number in this case. To see why this should be the case, just add the bit patterns for **OnOff** and **OffOn**.

Bitwise Addition: The OR Operator

The bitwise **OR** operator, also takes two whole number operands and returns a value of the same type. The returned value's bit pattern contains a 1 wherever corresponding bits for either operand or for both operands are 1, and a 0 when both operands have a 0 at that position.

Since a 1 in either operand is enough for a 1 in the result, the bitwise **OR** operator is useful for turning bits on. For example, you might use this operator to add elements to a collection.

The following program uses the bitwise **OR** operator to classify different animals into the appropriate groups. How would you write this program using sets?

Listing 10.7

```
(* Program to illustrate use of bitwise OR operator
   for building collections.
*)

PROGRAM OrDemo;
USES Windows;

CONST MaxNames = 15;
      MaxBits = 16;
      CRLF = #13#10;
(* Predefine some codes for animals. Note that the codes all
   have unique digits, because each is a different power of 2.
*)
      Possum   = $1;     Wolf     = $20;      Lynx     = $400;
      Wombat    = $2;      Fox       = $40;      Ocelot   = $800;
      Koala     = $4;      Whale     = $80;      Jaguar   = $1000;
      Numbat    = $8;      Porpoise = $100;      Cheetah  = $2000;
      Dog       = $10;     Narwhal  = $200;      Lion     = $4000;

TYPE SmStr = STRING [ 30];
     NameArray = ARRAY [ 1 .. MaxNames] OF SmStr;

VAR Marsupial, Canine,
    Feline, Cetacean,
    Carnivore, Mammal,
    Large, Small : INTEGER;
    Names : NameArray;

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END;  (* AwaitUser *)

  (* Initialize the array cells *)
  PROCEDURE InitNames ( VAR NameCells : NameArray;
                        Size : INTEGER);
  BEGIN
    NameCells [ 1] := 'possum';
    NameCells [ 2] := 'wombat';
    NameCells [ 3] := 'koala';
    NameCells [ 4] := 'numbat';
    NameCells [ 5] := 'dog';
    NameCells [ 6] := 'wolf';
    NameCells [ 7] := 'fox';
    NameCells [ 8] := 'whale';
    NameCells [ 9] := 'porpoise';
    NameCells [ 10] := 'narwhal';
    NameCells [ 11] := 'lynx';
    NameCells [ 12] := 'ocelot';
    NameCells [ 13] := 'jaguar';
    NameCells [ 14] := 'cheetah';
    NameCells [ 15] := 'lion';
  END;  (* InitNames *)

  (* Display the members of the specified class *)
  PROCEDURE DispMembers ( ClassName : SmStr;
                          Members : INTEGER);
  VAR Index, Count : INTEGER;
  BEGIN
    Index := 1;
    Count := 1;

    WRITELN ( CRLF, ClassName, ':');
    WHILE ( Index <= LION) AND ( Count < MaxBits) DO
    BEGIN
      IF ( Members AND Index <> 0) THEN
        WRITELN ( Names [ Count]);
      Index := Index * 2;
      INC ( Count);
    END; (* WHILE loop *)
  END; (* DispMembers *)


BEGIN (* Main Program *)
  InitNames ( Names, MaxNames);
  (* Assign animals to the classes.
     Note how the bitwise OR operator is used.
  *)
  Marsupial := Possum OR Wombat OR Koala OR Numbat;
  Canine := Dog OR Wolf OR Fox;
  Cetacean := Whale OR Porpoise OR Narwhal;
  Feline := Lynx OR Ocelot OR Jaguar OR Cheetah OR Lion;
  Carnivore := Canine OR Feline;
  Mammal := Carnivore OR Marsupial OR Cetacean;
  Large := Cetacean OR Feline OR Wombat OR Wolf;
  Small := Mammal AND NOT Large;

  DispMembers ( 'MARSUPIALS', Marsupial);
  DispMembers ( 'CETACEANS', Cetacean);
  READLN;
  DispMembers ( 'CANINES', Canine);
  DispMembers ( 'FELINES', Feline);
  READLN;
  DispMembers ( 'CARNIVORES', Carnivore);
  READLN;
  DispMembers ( 'MAMMALS', Mammal);
  READLN;
  DispMembers ( 'LARGE MAMMALS', large);
  DispMembers ( 'SMALL MAMMALS', small);
  AwaitUser;
END.
```

The program is easy to follow, despite its length. The constants are defined in order to make particular bit patterns available. Notice that the patterns are successive powers of 2. More importantly, all the constants have 1 in unique places --- that is, the 1s in the constants' values don't overlap.

To create a grouping, such as **Marsupial**, the program uses the bitwise **OR** operator to turn on the bits corresponding to the appropriate constants: **Possum**, Wombat, **Koala**, and **Numbat**.

To test your understanding of the bitwise **OR** operator, work through the calculations involved in assigning a value to **Feline**.

Let's see how **Small** is assigned a value. First, notice that **Mammal** comprises all the groups, as you can see from the following calculations. This collection will be used as a reference set from which **Large** animals will be excluded to leave **Small**. (The leftmost bit is not used in the examples, since there are only 15 animals. If the bit were used, you would need to work with **WORD** or **LONGINT** values.)

```
(* Build Mammal *)

   0000 0000 0111 0000 (Canine)
OR 0111 1100 0000 0000 (Feline)
----------------------------------
   0111 1100 0111 0000 (Carnivore)


   0111 1100 0111 0000 (Carnivore)
OR 0000 0000 0000 1111 (Marsupial)
OR 0000 0011 1000 0000 (Cetacean)
----------------------------------
   0111 1111 1111 1111 (Mammal)
```

Similarly, the following calculations yield the bit pattern for **NOT Large**:

```
   0000 0011 1000 0000 (Cetacean)
OR 0111 1100 0000 0000 (Feline)
OR 0000 0000 0000 0010 (Wombat)
OR 0000 0000 0010 0000 (Wolf)
---------------------------------
   0111 1111 1010 0010 (Large)

   1000 0000 0101 1101 (NOT Large)
```

The only animals that will be included in **Small** are those that are 1 in **NOT Large**.

Order of Precedence for the Bitwise Operators

The statement in which **Small** is assigned a value has two bitwise operators in succession. How do we know the order in which these should be applied?

The bitwise negation operator is a unary one. It has a high precedence, at the same level as the other unary operators you've seen so far.

The bitwise **AND** operator has a higher precedence that the bitwise **OR** operator. Recall that the logical counterparts of these operators were compared to multiplication and addition, respectively. Seen in this light, these operators again instantiate the precedence of multiplication over addition.

Bitwise Exclusion: The XOR Operator

In the preceding section, you saw how to use the bitwise **OR** operator to build groups of values. You also saw how to use the bitwise **AND** and the bitwise negation operators to help build such groups.

The bitwise **XOR** operator yields a 1 if exactly one of the corresponding operand bits is 1. That is, **XOR** yields 1 if either the left or the right bit is 1, *but not* if both bits are 1.

Toggling

The bitwise **XOR** operator is particularly useful for *toggling* bits --- that is, for turning a bit on when it is off, and turning it off when it is on. Toggling *always* changes the current value.

For example, suppose you decided on the basis of new information that a koala should be considered a large animal.

The easiest way to accomplish the desired change is to toggle the **Koala** bit in both **Large** and **Small**. The **Koala** bit is the one with value 4, which means the third bit from the right is involved.

In the current groupings, this bit will be 1 in **Small** and 0 in **Large**. The following two assignments would change both of these values --- thereby effecting the reclassification of a koala as large.

```
Large := Large XOR Koala;
Small := Small  XOR Koala;
```

If you changed your mind and decided to return to the original grouping, you could return to those values by simply repeating the preceding statements --- that is, by toggling the appropriate variables again.

You can toggle any bit pattern, regardless of whether it contains a single 1 bit or multiple bits that will each be toggled simultaneously.

The precedence of the bitwise **XOR** operator is the same as for the bitwise **OR** and other additive operators.

Shift Operators

Pascal also has operators for simply shifting a sequence of bits to the left or right by a specified number of positions. For example, when you shift to the left, the bits being vacated at the right end are filled with 0s. The bits at the left end simply drop off the variable and are discarded. Bits are not carried over into the next word.

The Left Shift Operator

The *left shift operator* (**SHL**), lets you move bits to the left. The operator takes two operands: an integral value whose bits will be shifted and a number specifying the number of places to shift --- that is, the number of places to be vacated at the right.

The following program lets you specify a left shift, and will then carry out this shift, one position at a time. The program will display the new value resulting from each left shift, until the desired number of places have been shifted.

Listing 10.8

```
(* Program to illustrate the left shift operator *)

PROGRAM SHLDemo;
USES Types;

VAR Index : INTEGER;
    ValToShift, NrPlaces, ShiftedVal : INTEGER;
    BinStr1, BinStr2 : STRING;

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* Await User *)

  (* Determine and return bit pattern for specified value. *)
  FUNCTION IntBits ( Value : WORD) : STRING;
  TYPE PattArray = ARRAY [ 0 .. 15] OF STRING [ 4];
  CONST (* Used to get individual hexadecimal digits *)
        Mask1 = $F;           Mask2 = $F0;
        Mask3 = $F00;         Mask4 = $F000;

  VAR  Patterns : PattArray;
       Temp, Piece : STRING;

     PROCEDURE InitPatterns;
     BEGIN
       Patterns [ 0] := '0000';
```

```
      Patterns [ 1] := '0001';
      Patterns [ 2] := '0010';
      Patterns [ 3] := '0011';
      Patterns [ 4] := '0100';
      Patterns [ 5] := '0101';
      Patterns [ 6] := '0110';
      Patterns [ 7] := '0111';
      Patterns [ 8] := '1000';
      Patterns [ 9] := '1001';
      Patterns [ 10] := '1010';
      Patterns [ 11] := '1011';
      Patterns [ 12] := '1100';
      Patterns [ 13] := '1101';
      Patterns [ 14] := '1110';
      Patterns [ 15] := '1111';
    END; (* InitPatterns *)

BEGIN  (* IntBits *)
  InitPatterns;
  Temp := '';

  Piece := Patterns [ Value AND Mask4 SHR 12];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask3 SHR 8];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask2 SHR 4];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask1];
  Temp := Temp + Piece;

  IntBits := Temp;
END; (* FN IntBits *)

(* Draw a line of a specified length on the screen.
   The line will be built of TheCh.
*)
PROCEDURE DrawLine ( Size : INTEGER;
                     TheCh : CHAR);
VAR Index : INTEGER;
BEGIN
```

```
    FOR Index := 1 TO Size DO
      WRITE ( TheCh);
    WRITELN;
  END; (* DrawLine *)

BEGIN
  WRITELN ( 'Left Shift demo program.');
  GetInt ( 'Shift what value?', ValToShift);
  GetInt ( '# places to shift?', NrPlaces);
  BinStr1 := IntBits ( ValToShift);
  WRITELN ( ValToShift : 5, ' : ', BinStr1);
  ShiftedVal := ValToShift;
  (* Shift bit by bit, displaying after each shift *)
  FOR Index := 1 TO NrPlaces DO
  BEGIN
    ShiftedVal := ShiftedVal SHL 1;
    BinStr2 := IntBits ( ShiftedVal);
    WRITELN ( 'Value SHL ', Index : 2, ' = ', ShiftedVal : 7,
              ' : ', BinStr2);
  END; (* FOR Index ... *)
  AwaitUser;
END.
```

For 255 as the starting value and 10 as the number of places to shift, the program produces the following output:

```
Left Shift demo program.
Shift what value? 255
# places to shift? 10
  255 : 0000000011111111
Value SHL  1 =     510 : 0000000111111110
Value SHL  2 =    1020 : 0000001111111100
Value SHL  3 =    2040 : 0000011111111000
Value SHL  4 =    4080 : 0000111111110000
Value SHL  5 =    8160 : 0001111111100000
Value SHL  6 =   16320 : 0011111111000000
Value SHL  7 =   32640 : 0111111110000000
Value SHL  8 =    -256 : 1111111100000000
Value SHL  9 =    -512 : 1111111000000000
Value SHL 10 =   -1024 : 1111110000000000
```

You can literally see the bits marching across the word. Notice that the bits on the right are always filled in with 0s for each shift.

Notice also that each new value is twice the starting value --- at least until you start dropping bits off the edge of the word. Shifting to the left by one position means multiplying by 2, just as adding a 0 at the end of a decimal means multiplying by 10.

Finally, notice what happens when the 1s reach the end of the word: they simply disappear. The function always returns a whole number value after carrying out the desired shift. If the bit pattern for this value has a 1 in the rightmost (sign) bit, the value will be negative, as you see for left shifts of 8 and more.

To avoid this problem of negative values, use **WORD** variables instead. Modify the program so that you are shifting bits in a **WORD**, which has no negative values. What values do you fid for the last few digits when you use **WORD** variables?

The Right Shift Operator

There is also a *right shift operator* (**SHR**) which shifts bits towards the right side of the number. Bits at the right end drop off the word and are discarded as bits are shifted from the left.

CodeWarrior Pascal fills the left bits with 0s, as the shift moves to the right. Right shift operations are not handled this way by all implementations or in all programming languages. The two common strategies are *logical shift* and *arithmetic shift*.

- In a logical shift, vacated bits are always replaced by 0. CodeWarrior Pascal uses a logical shift.
- In an arithmetic shift, vacated bits are filled with 1 if th sign bit was 1, and 0 otherwise.

The following program lets you watch the right shift operator at work.

Listing 10.9

```
(* Program to illustrate right shift operator *)

PROGRAM SHLDemo;
USES Types;

VAR Index : INTEGER;
    ValToShift, NrPlaces, ShiftedVal : INTEGER;
    BinStr1, BinStr2 : STRING;

  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* Await User *)

  (* Determine and return bit pattern for specified value. *)
  FUNCTION IntBits ( Value : WORD) : STRING;
  TYPE PattArray = ARRAY [ 0 .. 15] OF STRING [ 4];
  CONST (* Used to get individual hexadecimal digits *)
        Mask1 = $F;          Mask2 = $F0;
        Mask3 = $F00;        Mask4 = $F000;

  VAR  Patterns : PattArray;
       Temp, Piece : STRING;

     PROCEDURE InitPatterns;
     BEGIN
       Patterns [ 0] := '0000';
       Patterns [ 1] := '0001';
       Patterns [ 2] := '0010';
```

```
      Patterns [ 3] := '0011';
      Patterns [ 4] := '0100';
      Patterns [ 5] := '0101';
      Patterns [ 6] := '0110';
      Patterns [ 7] := '0111';
      Patterns [ 8] := '1000';
      Patterns [ 9] := '1001';
      Patterns [ 10] := '1010';
      Patterns [ 11] := '1011';
      Patterns [ 12] := '1100';
      Patterns [ 13] := '1101';
      Patterns [ 14] := '1110';
      Patterns [ 15] := '1111';
    END; (* InitPatterns *)

BEGIN  (* IntBits *)
  InitPatterns;
  Temp := '';

  Piece := Patterns [ Value AND Mask4 SHR 12];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask3 SHR 8];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask2 SHR 4];
  Temp := Temp + Piece + ' ';
  Piece := Patterns [ Value AND Mask1];
  Temp := Temp + Piece;

  IntBits := Temp;
END; (* FN IntBits *)

(* Draw a line of a specified length on the screen.
   The line will be built of TheCh.
*)
PROCEDURE DrawLine ( Size : INTEGER;
                     TheCh : CHAR);
VAR Index : INTEGER;
BEGIN
  FOR Index := 1 TO Size DO
    WRITE ( TheCh);
    WRITELN;
  END; (* DrawLine *)

BEGIN
  WRITELN ( 'Right Shift demo program.');
  GetInt ( 'Shift what value?', ValToShift);
  GetInt ( '# places to shift?', NrPlaces);
  BinStr1 := IntBits ( ValToShift);
  WRITELN ( ValToShift : 5, ' : ', BinStr1);
  ShiftedVal := ValToShift;
  (* Shift bit by bit; display bit pattern after each shift *)
  FOR Index := 1 TO NrPlaces DO
  BEGIN
    ShiftedVal := ShiftedVal SHR 1;
    BinStr2 := IntBits ( ShiftedVal);
    WRITELN ( 'Value SHR ', Index : 2, ' = ', ShiftedVal : 7,
              ' : ', BinStr2);
  END; (* FOR Index ... *)
  AwaitUser;
END.
```

Right Shifts and Powers of Two

You saw that shifting a bit pattern to the left amounts to multiplying by 2. Shifting to the right is essentially dividing by 2 --- at least in logical shifts. The output from the preceding program illustrates this, so if you haven't run it, do so now.

IntBits (Value : INTEGER) : STRING

Now that you've learned about the right shift operator, we can finally take a look at function **IntBits** to see how it accomplishes its task.

```
(* Determine and return bit pattern for specified value. *)
FUNCTION IntBits ( Value : WORD) : STRING;
CONST Patterns : ARRAY [ 0 .. 15] OF STRING [ 4] =
         ( '0000', '0001', '0010', '0011', '0100', '0101',
              '0110', '0111', '1000', '1001', '1010', '1011',
              '1100', '1101', '1110', '1111');

          (* Used to get individual hexadecimal digits *)
          Mask1 = $F;           Mask2 = $F0;
          Mask3 = $F00;         Mask4 = $F000;

  VAR Temp, Piece : STRING;
  BEGIN  (* IntBits *)
    Temp := '';

    Piece := Patterns [ Value AND Mask4 SHR 12];
    Temp := Temp + Piece + ' ';
    Piece := Patterns [ Value AND Mask3 SHR 8];
    Temp := Temp + Piece + ' ';
    Piece := Patterns [ Value AND Mask2 SHR 4];
    Temp := Temp + Piece + ' ';
    Piece := Patterns [ Value AND Mask1];
    Temp := Temp + Piece;

    IntBits := Temp;
  END; (* FN IntBits *)
```

The function builds the string **Temp** by adding successive values of **Piece** to the end of the string. Piece is assigned a value by getting a value between 0 and 15 --- the range covered by four bits --- and determining the bit pattern associated with that hexadecimal value. The bit pattern is stored in the appropriate cell of **Patterns**.

To get the values, **IntBits** uses four masks. Each mask takes one of the hexadecimal digits in an **INTEGER** representation. Thus, **Mask4** gets the leftmost hexadecimal digit of the value.

The result from this masking will be a very large value --- because it will have three trailing zeros. Obviously there is no **Pattern** cell corresponding to an index of over 4000. To bring the leftmost hexadecimal digit within range, it is shifted to the right by 12 places. This removes the trailing zeros. After this is accomplished, The **Pattern** cell corresponding to this shifted value will start the bit pattern string.

Similarly, **Mask3** is shifted, but only 8 bits, because it has only two trailing zeros. This sequence is repeated for each mask, until each digit's bit pattern has been added to the string. The spaces are added to make it easier to group bits.

Precedence of the Shift Operators

Since they produce the same effect as multiplication and division, these shift operators are grouped with other multiplicative operators, and have the same precedence as these operators.

Summary

This chapter covered a mix of topics, including units and bitwise operators. In the next chapter, we'll take the plunge into object-oriented programming.

11

Object-Oriented Programming in CodeWarrior Pascal

Over the past few years, object-oriented programming (OOP) has become the dominant programming strategy. OOP replaces the structured programming philosophy that guided programming for almost three decades.

OOP enables you to take advantage of some important properties that often hold for elements in a problem that your program is designed to solve. In this chapter and the next, you'll learn how to use object-oriented programming.

Traditional vs. Object-Oriented Programming

Traditional programming strategies are characterized succinctly in the title, *Algorithms + Data Structures = Programs* (1976, Englewood Cliffs, NJ: Prentice-Hall), of Niklaus Wirth's, book on structured programming.

In traditional programming, you design programs by determining what actions need to be performed and the most appropriate representation for the information to be transformed. The actions are systematized in algorithms and are implemented in procedures and functions. These routines are then applied to the program's data structures.

Example: Cataloging Written Materials

Suppose you want to start building a program to catalog written materials of various types. In such a project, you might want to keep track of author, title, and other information for books, articles, and miscellaneous publications.

The following program shows the beginnings of a traditional approach to building such a program. It contains procedures for carrying out actions such as initializing, displaying, and extending information about specified items. The information is stored in records.

Listing 11.1

```
PROGRAM PubDemo;
TYPE MedStr = STRING [ 80];

     Document = RECORD
                  Author, Title, Code : MedStr;
                  Year : INTEGER;
                END; (* Document *)

     Book = RECORD
              Author, Title,
              Code, Publisher : MedStr;
              Year : INTEGER;
              Price : REAL;
            END; (* Book *)

    Article = RECORD
                Author, Title, Code : MedStr;
                Year : INTEGER;
                Journal : MedStr;
                Volume, Page : INTEGER;
              END; (* Article *)

(* ************************************************** *)

  PROCEDURE InitDocument ( VAR Pub : Document;
                           Au, Ti : MedStr;
                           Yr : INTEGER);
  BEGIN
    WITH Pub DO
    BEGIN
      Author := Au;
      Title := Ti;
      Year := Yr;
      Code := '';
    END;
  END; (* InitDocument *)
```

```
PROCEDURE DescribeDocument ( Pub : Document);
BEGIN
  WITH Pub DO
    WRITELN ( Code, ': ', Author, ', ', Title,
              ' [', Year, ']');
END; (* DescribeDocument *)

PROCEDURE GetCode ( VAR TheCode : MedStr);
BEGIN
  WRITE ( 'Code? ');
  READLN ( TheCode);
END; (* GetCode *)

PROCEDURE CatalogDocument ( VAR Pub : Document);
BEGIN
  DescribeDocument ( Pub);
  GetCode ( Pub.Code);
END; (* CatalogDocument *)


PROCEDURE InitArticle ( VAR Art : Article;
                        Au, Ti, Jour : MedStr;
                        Yr, Vol, Pg : INTEGER);
BEGIN
  WITH Art DO
  BEGIN
    Author := Au;
    Title := Ti;
    Year := Yr;
    Code := '';
    Journal := Jour;
    Volume := Vol;
    Page := Pg;
  END;
END; (* InitArticle *)

PROCEDURE DescribeArticle ( Art : Article);
BEGIN
  WITH Art DO
```

```
    BEGIN
      WRITELN ( Code, ': ', Author, ', ', Title,
                ' [', Year, ']');
      WRITE ( Journal, ', ');
      WRITELN ( 'volume ', Volume, '; p. ', Page);
    END;
  END; (* DescribeArticle *)

  PROCEDURE CatalogArticle ( VAR Art : Article);
  BEGIN
    DescribeArticle ( Art);
    GetCode ( Art.Code);
  END; (* CatalogArticle *)

  PROCEDURE InitBook ( VAR TheBook : Book;
                       Au, Ti, Pub : MedStr;
                       Yr : INTEGER;
                       Pr : REAL);
  BEGIN
    WITH TheBook DO
    BEGIN
      Author := Au;
      Title := Ti;
      Code:= '';
      Year := Yr;
      Publisher := Pub;
      Price := Pr;
    END;
  END; (* InitBook *)

  PROCEDURE DescribeBook ( TheBook : Book);
  BEGIN
    WITH TheBook DO
    BEGIN
      WRITELN ( Code, ': ', Author, ', ', Title,
                ' [', Year, ']');
      WRITELN ( '$', Price : 1 : 2, ', ', Publisher);
    END;
  END; (* DescribeBook *)
  PROCEDURE CatalogBook ( VAR TheBook : Book);
  BEGIN
    DescribeBook ( TheBook);
    GetCode ( TheBook.Code);
  END; (* CatalogBook *)

VAR MyBk : Book;
    MyArt : Article;
    MyPub : Document;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
  InitDocument ( MyPub, 'Many Signers',
                  'Declaration of Independence', 1776);
  InitArticle ( MyArt, 'Mosteller, F.',
               'The Mystery of the Missing Corpus',
               'Psychometrika', 1958, 23, 279);
  InitBook  ( MyBk, 'Programming, O.O.', 'I, Object',
             'OOP Press', 1989, 22.95);
  WRITELN ( 'BOOK:');
  CatalogBook ( MyBk);
  DescribeBook ( MyBk);
  AwaitUser;
END.
```

The program illustrates some things about traditional programming strategies and stands in contrast to object-oriented solutions that you'll develop during the next two chapters.

First, look at the data structures used in the program. Three different record types are defined: **Document**, Article and **Book**. These records are all similar but not identical.

Most of the fields should be self-explanatory. The **Code** field provides a slot for storing an abbreviated name for the item. For example, many books and magazines use the first few letters of the author's name and the year of publication within square brackets, as in the following:

```
Most[58]
```

For each of the record types, you'll find corresponding procedures to do the following:

- Initialize a variable of the given type.
- Display the contents of such a variable.
- Modify certain fields of such a variable.

Another traditional approach would be to use a variant record to group all three kinds of documents within the same type definition. In that case, you would use a definition such as the following. Note that this variant record has an extra field — the tag field.

```
TYPE MedStr = STRING [ 80];
     PubType = ( doc, article, book);

     Document = RECORD
       Author, Title, Code : MedStr;
       Year : INTEGER;
       CASE WhatKind : PubType OF
         doc : ();
         article : ( Journal : MedStr;
                     Volume, Page : INTEGER);
         book : ( Publisher : MedStr;
                  Price : REAL);
     END; (* Document *)
```

With this approach you could use the same **Describe** and **Catalog** routines, regardless of the variable. However, you would need to add **CASE** statements to these routines, to handle the variants. Moreover, you would still need separate initialization routines for each of the variants, since the parameter list would contain too many unused arguments if you tried to use the same routine for all the variants.

An Object-Oriented Approach

In OOP, the approach is very different. Instead of focusing on the algorithms to be used and the data structures to which they should be applied, you first describe the entities involved in the program. The key in OOP is to find common features among entities, and to create data structures (“objects") that capture these common features. Other, related, entities can then share these features.

Think about how you would organize information about written materials. At the minimum, you would want to store author, title and year information. As part of your description of an object, you would also include information about how to use the object.

All written documents would have at least the three features mentioned; however, for certain types of documents you would store other information as well. Thus, a book is a special type of document; an article is also a type of document, but it differs from a book.

Object-oriented programming also lets you relate elements to each other. For example, in many problems and situations, related elements have features in common with each other. These features may be derived (inherited) from a common element, which is more general (that is, has fewer features) than the elements being compared.

In OOP, you can use the fact that books and articles are special cases of a document. In essence, once you've defined a document, you can define a book as “a document with some additional information associated with it." In principle, you can extend your definitions as far as necessary. For example, you could distinguish novels and textbooks as special cases of books (which are special cases of documents). You might further distinguish introductory and advanced textbooks, and so forth.

Similarly, poker, blackjack, and rummy are all card games. As such, they all use a deck, deal a fixed number of cards to each player, and so forth. These are some of the features that the three games might acquire from the more general category of card game. The three games differ in the way the cards are used by the players.

You can place such games in a hierarchy, with card games being a superordinate category under which you find the three types of card games. If you wish, you can even extend this hierarchy, both upward and downward. For example, card games are a specific example of games in general. Similarly, you can distinguish different types of poker games: 5-card draw, 7-card stud, baseball, and so forth.

Object-oriented programming lets you take advantage of the fact that elements in your programs can often be related in such a hierarchical fashion, with elements lower in the hierarchy having the features of elements higher in the hierarchy, as well as some features of their own. The data structures defined to represent such elements are known as *object types*, or *classes*.

An element below another in the hierarchy is said to be a *descendant* of the higher element. For example, a book and an article are both descendants of a document; blackjack is a descendant of card games. Conversely, an element above another in the hierarchy is said to be an *ancestor* of the lower element. Thus, a document is an ancestor of an article and of a book. A descendant is a special case of an ancestor, and has all the features of its ancestor, as well as additional features.

Inheritance

In object-oriented programming, you can build such acquired features directly into your object type definitions, using a property known as inheritance. *Inheritance* enables a descendant object type to have the features of its ancestor types. This capability makes certain programming tasks much easier and more convenient. You don't have to describe each feature explicitly for each element, since elements can inherit features from each other.

For example, suppose you're trying to represent information about animals. You may want to represent information about different types of animals — perhaps you want to differentiate among mammals, birds, and insects. These types represent subgroupings under the more general category of animals. For each of these groupings you'll have unique information to record; however, each also has more general characteristics that apply to all animals. Thus, mammals, birds, and insects share those features that characterize animals.

You may also want to differentiate within groupings. For example, you may want to store different information about primates (humans and apes) and cats, or about flies and ants. These subgroupings will have unique features, but will share more general features of their common ancestor groupings. Thus, both cats and primates are mammals. Consequently, they share the attributes associated with mammals. Since mammals are animals, cats and primates will also have those features.

As you've learned, a lower-level (that is, more specific) element is said to *inherit* features of more general elements. In addition, lower-level elements also have unique features. Thus, primates inherit the characteristics of a mammal and of an animal; humans are primates, which are mammals, which are animals. In the earlier example, poker, blackjack, and rummy are all card games; these, in turn, are a special case of a more general category: games.

Such a subdivision process can continue as many levels as required for your purposes. Standard Pascal has no suitable way of representing inheritance that spans multiple levels. Variant records are the most likely approach for representing such relationships. However, these are limited to one level of inheritance, since you can only have one variant part in a record.

In a variant record you could, for example, include both mammals and insects as variants within an animal record. You could not further distinguish primates and cats under mammals, however, without resorting to dirty programming tricks to fool the system.

CodeWarrior Pascal's object-oriented extensions enable you to represent such inheritance in a very clean way. You can define a high-level (general) element (such as animal). Once you've done this, you can define a lower-level element (such as insect) that is a special case of animal. By specifying an insect as a special case, you immediately associate an animal's features with this subdivision.

Associating Actions With Object Types

Standard programs are quite adequate for representing the features of problem elements. However, the standard programming solution can sometimes make the problem much more difficult than it has to be. One reason for this is the separation between data and routines.

This separation is unfortunate because, in real problems, actions are often associated in an integral way with particular elements. For example, dealing hands is an integral part of card games in our earlier example. The details of such an action will differ from game to game, however.

In object-oriented programming, such actions are included as part of an object type's definition. The actions are known as *methods* when associated with an object type. You'll see how to specify, define, and use methods after we look at how to define object types in CodeWarrior Pascal.

Representing Objects

Suppose you wanted to use an object-oriented program to catalog written materials. You might begin by defining a data type to represent a document — containing the minimal information that you expected to store about any kind of written material.

Representing a Document

In CodeWarrior Pascal, you can represent such an item by using the **OBJECT** type that his available in CodeWarrior Pascal's Object Pascal and in other object-oriented Pascal flavors. The following listing shows how to define a **Document** type.

```
TYPE MedStr = STRING [ 80];

     Document = OBJECT
       Author, Title, Code : MedStr;
       Year : INTEGER;
     END; (* Document *)
```

Notice that, except for the reserved word **OBJECT**, the syntax for the definition of **Document** is identical to a record definition. As you'll soon see, however, there are major differences between a record and an object type. For one thing, an object can only be defined in the global environment. That is, you can't define objects within a procedure or function.

An **OBJECT** definition is the third place where you can have an **END** without a corresponding **BEGIN**. (Recall that the other two are in a record definition and a **CASE** statement.)

Representing a Shape

As another example, suppose you wanted to create a program that uses geometric shapes. As a start, you might want to define a data type to represent such a shape. The following listing shows how to define a **Shape**, how to declare a variable of this type and also how to refer to a field within this variable.

Listing 11.2

```
(* Program showing how to define an object type,
   how to declare an object variable, and how to
   refer to a field within an object.
*)
PROGRAM ObjDemo;
TYPE SmStr = STRING [ 30];

     (* Define a shape as an object type with two features.
        Note how similar this syntax is to that for a record.
     *)
     Shape = OBJECT
               Name : SmStr;
               NrDimensions : INTEGER;
             END; (* Shape OBJECT *)

VAR MyShape : Shape;

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

BEGIN
```

```
   NEW ( MyShape);
   WRITELN ( 'MyShape      requires ', SIZEOF ( MyShape),
             ' bytes.');
   WRITELN ( 'MyShape.Name requires ', SIZEOF ( MyShape.Name),
             ' bytes.');
   AwaitUser;
END.
```

Once you've defined an object type, you can declare variables of that type. These variables are known as *instances* of the object, or simply as objects. Thus, the variable **MyShape** is an instance of a **Shape**. The syntax for declaring an object variable is the same as for any other type.

Internally, however, the declaration of an object variable is handled much differently — for reasons having to do with the properties of objects. Because of this, a program needs to get dynamic memory in which to store the object. The **NEW** procedure accomplishes this. This call will allocate the memory for the object and will return a reference to this memory. The reference is returned in the form of a handle — a number that identifies the object as a particular instance of its object class. (This use of **NEW** is actually an extension of standard Pascal's **NEW**.)

To access an object's fields, use the same technique as for records. Thus, to refer to a field, specify the variable's identifier, followed by a period, followed by the field name.

Representing Inheritance

An object type's fields can be inherited by other, more specialized object types. Thus, once you've defined a **Shape**, you can define different types of shapes. For example, you can define shapes with three sides (triangles), four sides (squares, diamonds, etc.), and round shapes (circles, ellipses). Each of these special shapes inherits all the features of a **Shape**, and each has some unique fields.

TIP: Only object types can inherit features from other object types. Inheritance is not possible for other Pascal data structures (such as records).

The following listing shows how to define descendant objects. **Shape** is the ancestor object type for the other three classes, which are all descendant object types. Again, try to predict the sizes of each variable before running the program.

Listing 11.3

```
(* Program to illustrate how to define and use
   descendant objects.
*)

PROGRAM ObjDemo2;

TYPE SmStr = STRING [ 30];

     (* Define a shape as an object type with two features.
        Note how similar this syntax is to that for a record.
     *)
     Shape = OBJECT
               Name : SmStr;
               NrDimensions : INTEGER;
             END; (* Shape OBJECT *)

     (* Define a triangle as a shape with 2 additional features.
        Note that a Shape's fields are not mentioned;
        the identifier Shape is used, however.
     *)
     Triangle = OBJECT ( Shape)
                  Angle1, Angle2 : REAL;
                END; (* Triangle OBJECT *)

     (* Define a quadrilateral as a shape with 3 additional
        features. Both this and triangle are more specialized
        instances of a Shape.
     *)
     Quadrilateral = OBJECT ( Shape)
                       Angle1, Angle2, Angle3 : REAL;
                     END; (* Quadrilateral OBJECT *)

     (* Define a Round as a shape with only one additional
        feature.
     *)
     Round = OBJECT ( Shape)
               MinRadius : REAL;
             END; (* Round OBJECT *)

VAR MyShape : Shape;
    MyTri : Triangle;
    MyQuad : Quadrilateral;
    MyRound : Round;

   (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

   (* Display information about the storage required for
      the type specified by Name. Note that the size must
      be passed into the procedure; it is not determined here.
  *)
  PROCEDURE DispSize ( Name : STRING;
                       Size : INTEGER);
  BEGIN
    WRITELN ( Name : 20, ' requires ', Size : 3, ' bytes.');
  END; (* DispSize *)

BEGIN
  DispSize ( 'MyShape', SIZEOF ( MyShape));
  DispSize ( 'MyShape.Name', SIZEOF ( MyShape.Name));
  WRITELN;

  DispSize ( 'MyTri', SIZEOF ( MyTri));
  DispSize ( 'MyTri.Name', SIZEOF ( MyTri.Name));
  WRITELN;

  DispSize ( 'MyQuad', SIZEOF ( MyQuad));
  DispSize ( 'MyQuad.Name', SIZEOF ( MyQuad.Name));
  WRITELN;
  DispSize ( 'MyRound', SIZEOF ( MyRound));
  DispSize ( 'MyRound.Name', SIZEOF ( MyRound.Name));
  WRITELN;

  AwaitUser;
END.
```

Similarly, the following definitions would let you describe books and articles as special types of documents. In the listing, **Book** and **Article** are descendants of **Document**, which is their ancestor.

Listing 11.4

```
PROGRAM DocDemo;

TYPE MedStr = STRING [ 80];

     Document = OBJECT
                  Author, Title, Code : MedStr;
                  Year : INTEGER;
                END; (* Document *)

     Book = OBJECT (Document)
              Publisher : MedStr;
              Price : REAL;
            END; (* Book *)

    Article = OBJECT (Document)
                Journal : MedStr;
                Volume, Page : INTEGER;
              END; (* Article *)

VAR MyDoc : Document;
    MyBk : Book;
    MyArt : Article;
    Ch : CHAR;
```

```
BEGIN
  NEW ( MyDoc);
  NEW ( MyBk);
  NEW ( MyArt);
   (* Code will eventually be included here *)
  WRITELN ( 'Sorry, nothing to report or display in this 
version.');
  WRITE ( 'Press ENTER to end program.');
  READLN ( Ch);

END.
```

The syntax of the definitions for **Book** and **Article** differs slightly from the definition for **Document**. Both **Book** and **Article** have the word **Document** in parentheses after the word **OBJECT**. In the listing, **Book** inherits features from **Document**, as does **Article**.

To define a descendant object type, such as a **Book**, simply include the name of the ancestor type within parentheses. This information is included after the word **OBJECT**. The fields to be associated with a **Book** are specified after this — just as in a record or in **Document**.

When you specify **Book** as a descendant of the **Document** type, you're describing a book as having four fields more than its definition would suggest. Although the fields are never mentioned explicitly in the definition of a **Book**, this type will have one **INTEGER** and three **MedStr** fields, in addition to its **Publisher** and **Price** fields.

Thus, you can specify **MyBk.Author** as a field for a **Book** variable, even though this field was not included explicitly in the **Book** definition. The syntax for specifying a field from an object type's ancestor indicates that **Document** is not nested within **Book**. Rather, a **Document**'s fields are part of the definition of a **Book**.

You can carry inheritance to further levels of descendants. For example, you could define a **Novel**, as in the following listing:

```
(* Define an object type that inherits the features of a Book
   which, in turn, inherits features of a Document.
   A novel variable will have seven fields:
   Author, Title, Code, Year (from Document);
   Publisher, Price (from Book); and
   Genre (unique to Novel).
*)
TYPE Novel = OBJECT ( Book)
                    Genre : MedStr;
                END; (* Novel *)
```

This definition says that a **Novel** has one explicit field, **Genre**. In addition, however, this type has all the fields of a **Book**. As a **Book**, a **Novel** is a **Document** with several additional fields. Thus, **Novel** has inherited fields from two other object types: directly from **Book** and indirectly (through **Book**) from **Document**.

Again, such inherited fields are all represented at the same level. For example, if **MyNovel** is of type **Novel**, then all of the following are valid fields:

```
MyNovel.Year;
MyNovel.Price;
MyNovel.Code;
MyNovel.Genre;
MyNovel.Author;
```

Inheritance can make your programming tasks easier, because you can create your objects incrementally. That is, you can define a general object type, and use it as part of more specific objects without explicitly needing to specify all the individual fields again. This is especially useful if you want to create new objects to use in your program, as you'll see in Chapter 12.

TIP: When you specify that an object type is a descendant of another type, the descendant class inherits all the features of the ancestor class, as well as of any classes inherited by the ancestor type or its ancestors.

Methods

Another feature of object-oriented programming is the ability to specify methods when defining an object. A *method* is a routine that is associated with a particular object type. This routine is available to any objects and descendant objects. In a sense, a method is part of an object type's definition.

TIP: A method is a procedure or function that is associated with an object type. You cannot associate methods with other Pascal data types (such as records).

The following listing expands the definition of a **Document**, and shows how to specify methods as part of an object type's definition.

```
TYPE MedStr = STRING [ 80];

     Document = OBJECT
       Author, Title, Code : MedStr;
       Year : INTEGER;
       (* Specify four methods associated with a Document *)
       PROCEDURE Init ( Au, Ti : MedStr;
                        Yr : INTEGER);
       PROCEDURE Describe;
       PROCEDURE GetCode;
       PROCEDURE Catalog;
     END; (* Document *)
```

This extended version of **Document** has several new components. The headings for four procedures are specified as part of **Document**'s definition. These methods are part of the definition of the object type within which they are specified. Thus, an element's features and actions are included within the definition of the type. This combination of features and methods within the same object type is known as *encapsulation*. We'll say that the four procedures are encapsulated within **Document**.

The procedure headings indicate that **Init** takes three arguments to do its work, while the other methods don't need any specific values when called. (Notice that the method actually changes four fields of a **Document**, but initializes one of them — **Code** — to a default value. This field is changed using one of the other methods.)

The implementations (that is, heading plus body) for these methods appear outside the type definition — generally in the section in which ordinary routines are defined.

The implementations for the encapsulated methods indicate just how closely methods are associated with their objects. Notice the names of the procedures.

```
(* Initialize fields of a Document to the specified values.
   Note that Init knows the names of Document's fields.
*)
PROCEDURE Document.Init ( Au, Ti : MedStr;
                          Yr : INTEGER);
```

```
BEGIN
  Author := Au;
  Title := Ti;
  Year := Yr;
  Code := '';   (* initialize to a default value *)
END; (* Document.Init *)

(* Display values of a Document's fields. *)
PROCEDURE Document.Describe;
BEGIN
 WRITELN ( Code, ': ', Author, ', ', Title, ' [', Year, ']');
END; (* Document.Describe *)

(* Get a value for the Code field from the user *)
PROCEDURE Document.GetCode;
BEGIN
  WRITE ( 'Code? ');
  READLN ( Code);
END; (* Document.GetCode *)

(* Catalog a particular element by getting a code for it.
   Note that the calls are to Describe and GetCode,
   rather than to Document.Describe, etc.
*)
PROCEDURE Document.Catalog;
BEGIN
  Describe;
  GetCode;
END; (* Document.Catalog *)
```

To indicate that the routines are a part of a **Document**'s definition, the method's name includes the name of the object type. Thus, the action is specified as if it were a field in the object type.

Document.Init assigns specified or default values to the object type's fields. **Document.Describe** displays the values of these fields. **Document.GetCode** gets a value for a specific field of a **Document**. Finally, **Document.Catalog** displays information about the object type, and then gets a value from the user.

A method automatically has direct access to the fields of its encapsulating object type. You don't need to use parameters to specify the actual fields within the object. Because an object's field names are known to a method associated with the object, you can't use the same identifiers in the method's parameter list. Doing so would be like using the same identifiers for parameters in the procedure heading and for local variables when defining a procedure.

TIP: A method has access to the fields of its encapsulating object type, and to any fields accessible to this type through inheritance.

Also, notice how the arguments passed to **Document.Init** are used. Values are assigned directly to **Author**, Title, and so forth. You don't have to specify the type name before the field. In fact, it is an error to do so.

Once you have defined these methods, you just need to call them with a suitable object as the variable whose fields are to be assigned values. The object used with the method should be of the same type as the one in the definition, but can also be a descendant of that type.

By making the method part of the object type's definition, the method becomes accessible to any instance of the object type. The next program shows how to refer to such an object. To call the **Init** method for your program's object variables, just specify the name of the **Document** variable you want to initialize. This name will be part of the call to **Init** in the program.

The following program lets you initialize a document variable as often as you want, and displays the values each time. Look at the definitions and the calls to methods associated with **Document**.

Listing 11.5

```
(* Program to illustrate how to specify, define and use
   methods.
*)

PROGRAM MethodDemo;
TYPE MedStr = STRING [ 80];

     Document = OBJECT
       Author, Title, Code : MedStr;
       Year : INTEGER;
       PROCEDURE Init ( Au, Ti : MedStr;
                        Yr : INTEGER);
       PROCEDURE Describe;
       PROCEDURE GetCode;
       PROCEDURE Catalog;
     END; (* Document *)

VAR MyDoc : Document;
```

```
    Writer, Name : MedStr;
    Year : INTEGER;

(* *************** Methods ************** *)


  PROCEDURE Document.Init ( Au, Ti : MedStr;
                            Yr : INTEGER);
  BEGIN
    Author := Au;
    Title := Ti;
    Year := Yr;
    Code := '';
  END; (* Document.Init *)

  PROCEDURE Document.Describe;
  BEGIN
    WRITELN ( Code, ': ', Author, ', ', Title, ' [', Year, ']');
  END; (* Document.Describe *)

  PROCEDURE Document.GetCode;
  BEGIN
    WRITE ( 'Code? ');
    READLN ( Code);
  END; (* Document.GetCode *)

  PROCEDURE Document.Catalog;
  BEGIN
    Describe;
    GetCode;
  END; (* Document.Catalog *)

(* *************** General-Purpose Routines ************** *)

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                        VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

  PROCEDURE GetInteger ( Message : STRING;
                         VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)

BEGIN
  NEW ( MyDoc);
  REPEAT
    GetString ( 'Author? (ENTER to stop)', Writer);
    IF Writer <> '' THEN
    BEGIN
      GetString ( 'Title', Name);
      GetInteger ( 'Year', Year);
      MyDoc.Init ( Writer, Name, Year);
      MyDoc.Catalog;
      MyDoc.Describe;
    END;
  UNTIL Writer = '';
  AwaitUser;
END.
```

Note that the program calls **MyDoc.Init**, even though the procedure definition was for **Document.Init**. When you actually call an object type's method, it will be to initialize an *instance* of the object. The **MyDoc** in the call indicates that initialization and display are to involve the fields of this particular document. The definition of **Document.Init** serves as a generic template for calls involving specific documents.

The **MedStr** fields of a **Document** are 80-character strings. The **GetString** procedure in program **MethodDemo** will get and return a full (255-character) string. If the compiler is doing strict **VAR**-string checking you will get a compiler error indicating a type mismatch.

To convince the compiler to let you get a **SmStr** with the **GetString** procedure, use the Copy strings using length byte option in the Pascal Compiler Project Settings. This relaxes the **VAR**-string checking.

Overriding Methods

When you define descendant object types, these types will also inherit any methods associated with the ancestor type. Thus, **Book**, and **Article** would both inherit **Init** and **Describe** methods from **Document**.

However, these inherited methods are insufficient when working with descendant types, which have more fields. In object-oriented programming, you can *override* an existing method by defining another one having the same name and associating this method with the descendant object.

The following program shows how to specify methods for descendant object types, and how to use these methods. The program lets you enter information to describe a **Document**, Book or **Article**. After you describe the instance, the program initializes and then displays the values of the instance's fields.

Listing 11.6

```
(*
  Program to illustrate how to specify methods to
  override methods inherited from ancestor object types.

  NOTE : Turn on the Copy String Using Length Byte option in
         the Pascal Compiler project settings --- to enable
         use of GetString for MedStr fields.
*)

PROGRAM OverrideDemo;

TYPE MedStr = STRING [ 80];

     Document = OBJECT
                  Author, Title, Code : MedStr;
                  Year : INTEGER;
                  PROCEDURE Init ( Au, Ti : MedStr;
                                   Yr : INTEGER);
                  PROCEDURE Describe;
                  PROCEDURE GetCode;
                  PROCEDURE Catalog;
                END; (* Document *)

     Book = OBJECT (Document)
              Publisher : MedStr;
              Price : REAL;
              PROCEDURE BkInit ( Au, Ti, Pub : MedStr;
                               Yr : INTEGER;
                               Pr : REAL);
              PROCEDURE Describe; OVERRIDE;
(*              PROCEDURE Catalog;  NOT needeed *)
            END; (* Book *)

    Article = OBJECT (Document)
                Journal : MedStr;
                Volume, Page : INTEGER;
                PROCEDURE ArtInit ( Au, Ti, Jour : MedStr;
                                 Yr, Vol, Pg : INTEGER);
                PROCEDURE Describe; OVERRIDE;
(*                PROCEDURE Catalog; NOT needed *)
              END; (* Article *)

VAR MyDoc : Document;
    MyBk : Book;
    MyArt : Article;
    Writer, Name, Pub, Jour : STRING;
    Choice, Year, Vol, Pg : INTEGER;
    Price : REAL;

(* *************** Methods *************** *)

  PROCEDURE Document.Init ( Au, Ti : MedStr;
                            Yr : INTEGER);
  BEGIN
    Author := Au;
    Title := Ti;
    Year := Yr;
    Code := '';
  END; (* Document.Init *)

  PROCEDURE Document.Describe;
  BEGIN
    WRITELN ( Code, ': ', Author, ', ', Title, ' [', Year, ']');
  END; (* Document.Describe *)

  PROCEDURE Document.GetCode;
  BEGIN
    WRITE ( 'Code? ');
    READLN ( Code);
  END; (* Document.GetCode *)

  PROCEDURE Document.Catalog;
  BEGIN
    Describe;
    GetCode;
  END; (* Document.Catalog *)


  PROCEDURE Article.ArtInit ( Au, Ti, Jour : MedStr;
                              Yr, Vol, Pg : INTEGER);
  BEGIN
    Init ( Au, Ti, Yr);
    Journal := Jour;
    Volume := Vol;
    Page := Pg;
  END; (* Article.Init *)

  PROCEDURE Article.Describe;
  BEGIN
    INHERITED Describe;
    WRITE ( Journal, ', ');
    WRITELN ( 'volume ', Volume, '; p. ', Page);
  END; (* Article.Describe *)

  PROCEDURE Book.BkInit ( Au, Ti, Pub : MedStr;
                          Yr : INTEGER;
```

```
                       Pr : REAL);
  BEGIN
    INHERITED Init ( Au, Ti, Yr);
    Publisher := Pub;
    Price := Pr;
  END; (* Book.Init *)

  PROCEDURE Book.Describe;
  BEGIN
    INHERITED Describe;
    WRITELN ( '$', Price : 1 : 2, ', ', Publisher);
  END; (* Book.Describe *)

(* *************** General-Purpose Routines *************** *)

  (* Wait for user to presses ENTER *)
  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue. ');
    READLN ( Ch);
  END; (* AwaitUser *)

  PROCEDURE GetString ( Message : STRING;
                       VAR Value : STRING);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetString *)

  PROCEDURE GetInteger ( Message : STRING;
                        VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInteger *)


  PROCEDURE GetReal ( Message : STRING;
                     VAR Value : REAL);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN
  NEW ( MyDoc);
  NEW ( MyBk);
  NEW ( MyArt);
  GetInteger ( '0) Quit; 1) Doc; 2) Book; 3) Article', Choice);
  WHILE Choice <> 0 DO
  BEGIN
    GetString ( 'Author?', Writer);
    GetString ( 'Title', Name);
    GetInteger ( 'Year', Year);
    CASE Choice OF
      1 :
        BEGIN
          MyDoc.Init ( Writer, Name, Year);
          MyDoc.Catalog;
          MyDoc.Describe;
        END;
      2 :
        BEGIN
          GetString ( 'Publisher', Pub);
          GetReal ( 'Price', Price);
          MyBk.BkInit ( Writer, Name, Pub, Year, Price);
          MyBk.Catalog;
          MyBk.Describe;
        END;
      3 :
        BEGIN
          GetString ( 'Journal', Jour);
          GetInteger ( 'Volume', Vol);
          GetInteger ( 'Page', Pg);
          MyArt.ArtInit ( Writer, Name, Jour, Year, Vol, Pg);
          MyArt.Catalog;
          MyArt.Describe;
        END;
    END;
    GetInteger ('0) Quit; 1) Doc; 2) Book; 3) Article', Choice);
  END; (* WHILE Loop *)
  AwaitUser;
END.
```

As descendants of **Document**, the **Book** and **Article** types inherit **Document.Init** and **Document.Describe**. But these descendants also have their own initialization methods (**Article.ArtInit** and **Book.BkInit**), since they have additional fields. **Book** and **Article** can, however, call the **Document.Init** method.

When you specify new **Describe** methods within **Book** and within **Article**, these will override the method inherited from **Document**. However, when the main program calls **MyArt.ArtInit**, this method can call **Document.Init**. The **OVERRIDE** after the methods declarations indicate that the methods are new.

Notice that **Book.BkInit** does call **Document.Init** to initialize the fields inherited from that ancestor object type. This is considered better programming practice than initializing **Author**, Title and **Year** explicitly in **Book.Init**.

When **Document.Init** is called in this way, the program knows what variable is meant. This is because the call to **Book.Init** has already specified a variable. Within the context of this method, all activity is assumed to apply to the instance specified in the call.

Book.BkInit and **Document.Init** take a different number of arguments, which is why **Book** needs a separate method. Common methods must all have the same number and types of arguments.

You can specify methods for descendants of descendants, and so forth. Each method that has the same name as a method already defined in an ancestor object type can override the method of the same name in the more general object type.

Thus, when the program encounters a call to a method, it searches the local environment of the object specified in the call. If a method with that name is found, this method is executed. If no such method is associated directly with the object, the program checks the ancestor object for such a method. This process continues until a method with the specified name is found. The first matching method is executed — provided that the call has the appropriate number and types of parameters.

Assignments Between Different Object Types

When you're dealing with objects, you can actually assign one object to another even if these are not exactly the same type. There are restrictions, however.

You can assign an instance of a descendant object to an instance of an ancestor object. For example, if you have a **Document** and a **Book** variable, you can assign the latter to the former in a single assignment statement. The converse (assigning a **Document** instance to a **Book** variable) is not allowed.

TIP: You can assign an instance of a descendant object to a variable of the ancestor's type, but you cannot assign an ancestor instance to a variable of the descendant's type.

Similarly, you could assign a variable of type **Triangle** to one of type **Shape**, but you can't assign the ancestor object to a variable of a descendant type. The following program illustrates this.

Listing 11.7

```
(* Program to illustrate how to assign instances of
   different, but related object types in a single statement.
*)

PROGRAM AssignmentDemo;

CONST EmptyString = '';
TYPE SmStr = STRING [ 30];

     (* Define a Shape as an object type with two features
        and with two associated methods.
     *)
     Shape = OBJECT
               Name : SmStr;
               NrDimensions : INTEGER;
               PROCEDURE Init ( NewName : SmStr;
                                Dims : INTEGER);
               PROCEDURE Disp;
             END; (* Shape OBJECT *)

     (* Define a Triangle as a descendant of a Shape.
        The Init and Disp methods specified for Triangle
        will override the ones inherited from Shape.
     *)
     Triangle = OBJECT ( Shape)
                  Angle1, Angle2 : REAL;
                  PROCEDURE TriInit ( NewName : SmStr;
                                      Dims : INTEGER;
                                      A1, A2 : REAL);
                  PROCEDURE Disp; OVERRIDE;
                END; (* Triangle OBJECT *)

VAR MyShape : Shape;
    MyTri : Triangle;
    StrVal : SmStr;
    AngleVal1, AngleVal2 : REAL;
    DimVal : INTEGER;

  (* Initialize the fields of a Shape variable with the
     values passed in NewName and Dims.
     The variable will be the one whose name replaces
     Shape when the method is called.
  *)
  PROCEDURE Shape.Init ( NewName : SmStr;
                         Dims : INTEGER);
  BEGIN
    Name := NewName;
    NrDimensions := Dims;
  END; (* Shape.Init *)

  (* Display the fields of a Shape. *)
  PROCEDURE Shape.Disp;
  BEGIN
    WRITELN ( Name, ': ', NrDimensions, ' dimensions');
  END; (* Shape.Disp *)

  (* Initialize the fields of a Triangle. This method takes four
     arguments: the first two initialize the "Shape" fields, and
     the last two initialize fields unique to a Triangle.
     NOTE that the method actually calls the method Triangle.Init
     is overriding.
  *)
  PROCEDURE Triangle.TriInit ( NewName : SmStr;
                               Dims : INTEGER;
```

```
                             A1, A2 : REAL);
BEGIN
  (* Call the ancestor method to initialize the ancestor's
     fields.
  *)
  INHERITED Init ( NewName, Dims);
  Angle1 := A1;
  Angle2 := A2;
END;  (* Triangle.Init *)

(* Display the contents of a Triangle variable.
   Note that Shape.Disp is called.
*)
PROCEDURE Triangle.Disp;
BEGIN
  INHERITED Disp;
  WRITELN ( 'ANGLES: 1 = ', Angle1 : 5 : 2,
            '; 2 = ', Angle2 : 5 : 2,
            '; 3 = ', 180 - Angle1 - Angle2 : 5 : 2,
            ' deg');
END;   (* Triangle.Disp *)

(* Wait for user to presses ENTER *)
PROCEDURE AwaitUser;
VAR Ch : CHAR;
BEGIN
  WRITE ( 'Press ENTER to continue. ');
  READLN ( Ch);
END; (* AwaitUser *)

PROCEDURE GetString ( Message : STRING;
                      VAR Value : STRING);
BEGIN
  WRITE ( Message, ' ');
  READLN ( Value);
END; (* GetString *)

PROCEDURE GetReal ( Message : STRING;
                    VAR Value : REAL);
BEGIN
```

```
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetReal *)

BEGIN  (* Main Program *)
  NEW ( MyShape);
  NEW ( MyTri);
  GetString ( 'Shape Name? (ENTER to quit)', StrVal);
  WHILE ( StrVal <> EmptyString) DO
  BEGIN
    GetReal ( 'Angle 1', AngleVal1);
    GetReal ( 'Angle 2', AngleVal2);
    (* To initialize a Triangle, use the method
       appropriate to this descendant type.
    *)
    MyTri.TriInit ( StrVal, 2, AngleVal1, AngleVal2);
    MyTri.Disp;
    WRITELN;
    WRITELN ( 'Assigning MyTri to MyShape:');
    MyShape := MyTri;
    MyShape.Disp;
    WRITELN ( '*******************************************');
    GetString ( 'Shape Name? (ENTER to quit)', StrVal);
  END;
  AwaitUser;
END.
```

In this program, **MyTri** is assigned to **MyShape**. The compiler does not complain about this assignment. On the other hand, if you reversed the two variables — to assign **MyShape** to **MyTri** — the compiler would respond with a type mismatch error.

Let's look carefully at what's going on here. First, note that a **Triangle** is actually larger (requires more storage) than a **Shape**. So how can the compiler make this assignment? In fact, the extra components are simply discarded, and only the fields that are part of **MyShape** are assigned.

Why can't you assign a smaller variable to a larger one in this case? If you were to assign **MyShape** to **MyTri**, only some of the fields in **MyTri** would be assigned values, leaving the remaining components uninitialized. Since you can get into lots of trouble if you have an uninitialized variable in a program, such an action is not allowed.

The Invisible Self *Parameter*

Let's look back at the **OverrideDemo** program earlier in the chapter (**11-6**). Notice that there are three versions of **Describe** — one for each of the three object types used in the program. The two descendant versions each include an **OVERRIDE** statement. There is only one copy each of **Catalog** and **GetCode**, however.

It should be clear why only one version of **GetCode** is needed: The method does exactly the same thing for any of the objects. This routine modifies a field (**Code**) that is inherited from the **Document** type. If the **Code** field were found only in **Book** but not in **Document**, you would need a **GetCode** method for **Book** but not for the other object types.

Catalog makes the same calls for each object type — to **Describe** and to **GetCode** — but the methods involved are not necessarily the same. How does **MyBk.Catalog** know to call the **Describe** method for **Book**, rather than the one for **Document**? The answer has to do with the way in which the system decides what object you're working with. When you call a method, an invisible parameter, **Self**, is passed to the method. This parameter represents the object instance involved in the procedure call.

For example, if you call **MyBk.Describe**, the **Self** parameter would refer to **MyBk**. Similarly, when there is a call to **Describe** in the body of **MyBk.Catalog**, this is actually interpreted as a call to **Self.Describe**, which is interpreted as **Book.Describe** in that instance.

If no such method is found, the system begins searching ancestor objects for a method called **Describe**. In the example, the method would be found associated with **Document**. That version of **Describe** would be executed. (Note that the ancestor's method could be used because a descendant is compatible with an ancestor, as you saw in the preceding section.)

Summary

In this chapter, you've been introduced to the major features of objects and to object-oriented programming. The major features discussed in this chapter concerned the ability to associate objects and methods within an object definition, and the ability to inherit features from ancestor objects.

In the next chapter, you'll learn about some other advantages provided by object-oriented programming — that make it possible for you to build on and add to existing objects — even if you don't have access to the source code for the original objects. Moreover, object-oriented programming lets you design your objects and methods in a way that will make it easy to extend them at a later point of you wish.

12

More on Object-Oriented Programming

In the last chapter, you learned a bit about the major features of object-oriented programming. In particular, you learned how to define object types, and how to declare methods within these types. You also learned how to use object variables and methods.

The principles of inheritance, encapsulation, and overriding all help to differentiate an object oriented approach from traditional programming. In OOP, you can pass on features from one object type to another. The descendant type (which inherits the features) also has access to any methods defined for the ancestor type or accessible to it.

The ability to override methods, in order to apply methods better suited to a descendant type, is one feature that enables you to create a family of methods for accomplishing the same type of task with different objects.

Overriding enables you to create as many objects as you need to accomplish your tasks, and to carry out the same kinds of actions on the objects you've declared. Another important consequence of overriding is to make it possible to delay certain decisions until runtime. In particular, the decision as to which version of an overridden method to use can be delayed until runtime. This is known as late binding.

Binding

When the compiler processes your source file, it creates a code file that contains the program's routines in a code segment, and variables and constants in a data segment. Thus, both routines and variables will have addresses associated with them.

When a either procedure or the program calls another routine, the location of the called routine must be provided to the caller. This process is called *binding*. Generally, binding occurs when the program is compiled and linked. This is known as *early binding*. Standard Pascal, and most other programming languages (such as C, Modula-2, and BASIC) use early binding.

In the example in Chapter 11, early binding means that the address of procedure **Document.Describe** is provided during compilation for the call to **Describe** in **Document.Catalog**. Thus, the method being called is determined before the program ever executes.

It's possible, however, to delay the binding until run-time. This is known as *late binding*. In late binding, the program does not need to decide what routine to call until the program is actually executing.

Essentially, early binding says, “when you execute, you are to carry out the instructions contained in this specific routine." In contrast, late binding says, “when you execute, you'll be told exactly which routine's instructions you are to carry out." In late binding, the decision as to which **Describe** routine to execute can be made just before it's necessary to execute it.

This means you can, in fact, use **Document.Catalog** for any descendant objects that don't have their own **Catalog** methods. When **Document.Catalog** is called, the system will determine what object's **Describe** method is called. In order to do this, however, descendant methods must be able to override their ancestors.

CodeWarrior Pascal implements late binding by means of the **OVERRIDE** statement. Through overriding, a method (such as **Describe**) is made to represent any of several methods — all of which have the same name, and each of which is associated with a different object type.

Polymorphism

A collection of methods that use **OVERRIDE** illustrates what is known as *polymorphism*, which refers to the fact that the same action (such as describing) can take different forms. With respect to the program, it also means that a single method name (such as **Describe**) can be instantiated in any of several ways. (Polymorphism comes from two Greek terms, meaning “many" and “shape.")

Flexibility

A major benefit of polymorphism is the flexibility it offers for creating objects of all sorts, simply by deriving them from existing objects. In this chapter, we'll look at at few examples to illustrate this. You'll also find numerous suggestions for how you can further develop the ideas and objects presented in this chapter. So let's get going!

Beginning with a Collection of Numbers

The example listing in this section introduces an object type that will turn out to be amazingly fertile as a starting point for descendants. Before we create the program, let's build the data structure we'll be using in most of the chapter.

We're going to define an object for creating and administering a collection of numbers. The data structure for our number group (**NrGrp**) is:

```
CONST MAX_VALUES = 500;

TYPE NrGrp = OBJECT
       NrVals : INTEGER;    (* # numbers in the array *)
       Nrs : ARRAY [ 1 .. MAX_VALUES] OF INTEGER;  (* number
container *)
        (* initialize the array and the counter *)
       PROCEDURE Init;
        (*
           Returns the number of values in the array
           --- ie, the value of NrVals
        *)
       FUNCTION ShowSize : INTEGER;
        (* Write a value to the next available cell *)
       PROCEDURE PutVal ( TheVal : INTEGER);
      (* Retrieve a value from a specified cell in the array *)
       FUNCTION GetVal ( Index : INTEGER) : INTEGER;
      (* Search through the array cells for a specified value
*)
       FUNCTION FindVal ( TheVal : INTEGER) : INTEGER;
     END;
```

This object has two fields and five methods. One of the fields — **Nrs** — is an array that can be used to hold up to **MAX_VALUES** integers. The other field is simply a counter that indicates the number of values currently in the array. The associated methods are used to build, check, and report on the numbers in the array.

Sounds simple enough, doesn't it? It is! Despite that, however, you can develop this object in dozens of ways. you can end up for example, with a statistics package or a blackjack game, a Monte Carlo simulation or a package some unexpected results in probability. We'll do some of these things — but on a very limited scale. If the possibilities catch your fancy, you can develop such things without too much work.

NrGrp ***Methods***

The method definitions are as follows:

```
   (* Initialize the object's data elements. *)
   PROCEDURE NrGrp.Init;
   VAR Index : INTEGER;
   BEGIN
     FOR Index := 1 TO MAX_VALUES DO
       Nrs [ Index] := 0;
     NrVals := 0;
   END; (* NrGrp.Init *)

  (* Return the number of elements currently in the array *)
  FUNCTION NrGrp.ShowSize : INTEGER;
  BEGIN
    ShowSize := NrVals;
  END; (* NrGrp.ShowSize *)

  (* Add a value to the array *)
  PROCEDURE NrGrp.PutVal ( TheVal : INTEGER);
  BEGIN
    IF NrVals <= MAX_VALUES THEN
    BEGIN
      INC ( NrVals);
      Nrs [ NrVals] := TheVal;
    END;
  END;  (* NrGrp.PutVal *)

  (* Retrieve the contents of the specified cell *)
  FUNCTION NrGrp.GetVal ( Index : INTEGER) : INTEGER;
  CONST ERROR_VAL = -32768;
  BEGIN
   (* Make sure the requested value is within the valid range
*)
    IF ( Index >= 1) AND ( Index <= NrVals) THEN
      GetVal := Nrs [ Index]
    ELSE
      GetVal := ERROR_VAL;
  END;  (* FUNCTION NrGrp.GetVal *)

  (* Search the array for a specified data value. *)
  FUNCTION NrGrp.FindVal ( TheVal : INTEGER) : INTEGER;
  VAR Index : INTEGER;
      Found : BOOLEAN;
  BEGIN
    Found := FALSE;
    Index := 1;
    WHILE ( Index <= NrVals) AND ( NOT Found) DO
    BEGIN
      IF Nrs [ Index] = TheVal THEN
```

```
          Found := TRUE
        ELSE
          INC ( Index);
      END;
      IF Found THEN
        FindVal := Index
      ELSE
        FindVal := 0;
    END; (* FUNCTION NrGrp.FindVal *)
```

Even these few routines afford lots of possibilities. With a convenient random number generator, you can start doing interesting things right from the start. In the example program, we'll fill the **NrGrp** array with values and we'll look at some details of these numbers.

In particular, we'll count the number of odd and even values. These values should be about equal, although there is room for variation with a sample as small as 500.

The program also looks at a result that I've always considered one of the most interesting findings related to probability. **OddLead**, EvenLead, and **Ties** in procedure **SampleCells** collects this information. For example, **OddLead** counts the number of trials during which there have been more odd than even values up to that point.

To see what this value means, consider the series 2, 4, 6, 8, 1, 3, 5, 10, 7, 9. For this series, **OddLead** would be zero, **EvenLead** would be 9, and **Ties** would be 1. Thus, even though there were just as many odd as even values, odd was playing catchup during the entire series. It turns out that this is almost always the case. In thousands of trials, one value will be leading about 80-90 percent of the time — even though there might be an exactly even number of both kinds of outcomes. In the world of probability, catching up is very difficult. For those of you who can find life lessons in programming books, keep this in mind the next time you're thinking of recouping losses you've already incurred.

Here's program **12-1.PAS**.

Listing 12.1

```
(* Program to illustrate definition and use of a NrGrp object *)

PROGRAM NrGrpDemo;
USES Windows;

CONST NR_SAMPLES = 100;
      MAX_VALUES = 500;
      PER_LINE = 10;
```

```
TYPE NrGrp = OBJECT
       NrVals : INTEGER;  (* # numbers in the array *)
       Nrs : ARRAY [ 1 .. MAX_VALUES] OF INTEGER; (* number
container *)
       (* initialize the array and the counter *)
       PROCEDURE Init;
       (*
          Returns the # of values in the array
          --- ie, the value of NrVals
       *)
       FUNCTION ShowSize : INTEGER;
       (* Write a value to the next available cell *)
       PROCEDURE PutVal ( TheVal : INTEGER);
       (* Retrieve a value from a specified cell in the array *)
       FUNCTION GetVal ( Index : INTEGER) : INTEGER;
       (* Search through the array cells for a specified value *)
       FUNCTION FindVal ( TheVal : INTEGER) : INTEGER;
     END;  (* NrGrp OBJECT *)

VAR MyNrs : NrGrp;
    Index, HowManyCells, RandSeed : INTEGER;

(* ************************************* *)
(*       Method Definitions              *)
(* ************************************* *)

   (* Initialize the object's data elements *)
   PROCEDURE NrGrp.Init;
   VAR Index : INTEGER;
   BEGIN
     FOR Index := 1 TO MAX_VALUES DO
       Nrs [ Index] := 0;
     NrVals := 0;
   END; (* NrGrp.Init *)

  (* Return the number of elements currently in the array *)
  FUNCTION NrGrp.ShowSize : INTEGER;
  BEGIN
    ShowSize := NrVals;
  END; (* NrGrp.ShowSize *)
```

```
(* Add a value to the array *)
PROCEDURE NrGrp.PutVal ( TheVal : INTEGER);
BEGIN
  IF NrVals <= MAX_VALUES THEN
  BEGIN
    INC ( NrVals);
    Nrs [ NrVals] := TheVal;
  END;
END;  (* NrGrp.PutVal *)

(* Retrieve the contents of the specified cell *)
FUNCTION NrGrp.GetVal ( Index : INTEGER) : INTEGER;
CONST ERROR_VAL = -32768;
BEGIN
  (* Nake sure the requested value is within the valid range *)
  IF ( Index >= 1) AND ( Index <= NrVals) THEN
    GetVal := Nrs [ Index]
  ELSE
    GetVal := ERROR_VAL;
END;  (* FUNCTION NrGrp.GetVal *)

(* Search the array for a specified data value *)
FUNCTION NrGrp.FindVal ( TheVal : INTEGER) : INTEGER;
VAR Index : INTEGER;
    Found : BOOLEAN;
BEGIN
  Found := FALSE;
  Index := 1;
  WHILE ( Index <= NrVals) AND ( NOT Found) DO
  BEGIN
    IF Nrs [ Index] = TheVal THEN
      Found := TRUE
    ELSE
      INC ( Index);
  END;
  IF Found THEN
    FindVal := Index
  ELSE
    FindVal := 0;
  END; (* FUNCTION NrGrp.FindVal *)

(* ************************************* *)
(*        Auxiliary Routines             *)
(* ************************************* *)

  (* A shell within which you can put whatever random
     number generator you might want to use.
     In this version, it just calls the RAND function.
  *)
  FUNCTION PlainRand : INTEGER;
  BEGIN
    PlainRand := RAND;
  END;  (* FUNCTION PLainRand *)

  (* Get a number from the user *)
  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

(* ************************************* *)
(*     Program-Specific Routines         *)
(* ************************************* *)

  (* Examine the values of a sample of cells, and report
     some information about the properties of these values.
     NOTE: OddLead represents how often NrOdd was greater than
           NrEven when we checked; EvenLead represents the # of
           times NrEven was in the lead; Ties represents the # of
           times NrEvn and NrOdd were equal when we checked.
  *)
  PROCEDURE SampleCells;
  VAR Index,
      Result, WhichCell,
      NrOdd, NrEven,
      OddLead, EvenLead, Ties : INTEGER;
  BEGIN
    NrOdd := 0;  (* # of odd values in the sample *)
    NrEven := 0; (* # of even values in the sample *)

    OddLead := 0; EvenLead := 0; Ties := 0;
    FOR Index := 1 TO NR_SAMPLES DO
    BEGIN
      (* select a cell, to see its value *)
      WhichCell := PlainRand MOD MAX_VALUES;
      Result := MyNrs.GetVal ( WhichCell);
      IF ODD ( Result) THEN
        INC ( NrOdd)
      ELSE
        INC ( NrEven);
      IF NrOdd > NrEven THEN
        INC ( OddLead)
      ELSE IF NrEven > NrOdd THEN
        INC ( EvenLead)
      ELSE
        INC ( Ties);
    END;
    WRITELN ( 'Of ', NR_SAMPLES : 0, ' cells checked, ',
                     NrOdd : 0, ' had odd values, and ',
                     NrEven : 0, ' had even values.');
    WRITELN ( 'During the sampling, Odd was leading ', OddLead :
0,
              ' times, ');
    WRITELN ( 'even was leading ', EvenLead : 0,
              ' times, and there were ', Ties : 0, ' ties.');
  END; (* SampleCells *)

  PROCEDURE ShowNumbers (NrCells : INTEGER);
  VAR Index : INTEGER;
  BEGIN
```

```
    IF NrCells > MAX_VALUES THEN
    BEGIN
      WRITELN ( 'There are only ', MAX_VALUES:0,
                ' cells to see.');
      NrCells := MAX_VALUES;
    END;
    WRITELN ( 'The values in the first ', NrCells:0,
              ' cells are shown, 10 values per line.');
    FOR Index := 1 TO NrCells DO
    BEGIN
      WRITE ( MyNrs.GetVal ( Index) : 7);
      IF ( Index MOD PER_LINE) = 0  THEN
        WRITELN;
    END;
  END;  (* ShowNumbers *)

BEGIN  (* Main Program *)
  GetInt ( 'Seed for the Random Number Generator?', RandSeed);
  SRAND ( RandSeed);
  (* Allocate storage for the object *)
  NEW ( MyNrs);
  MyNrs.Init;   (* Initialize it *)

  (* Feed some values to the number group array *)
  FOR Index := 1 TO MAX_VALUES DO
    MyNrs.PutVal ( PlainRand);

  SampleCells;

  GetInt ( 'Display how many cell values?', HowManyCells);
  ShowNumbers ( HowManyCells);

  AwaitUser;
END.
```

There are a few other things to notice about the program:

- First, we've apparently managed to sneak a type definition into a type definition. Notice the base type of **Nrs** in the definition of **NrGrp**. This type is one you would ordinarily expect to see as a type definition. Pascal will let you get away with such "on-the-fly" type definitions in certain cases. The price you pay is that you can't create any other variables of this type. The **Nrs** field in a **NrGrp** object is the only **ARRAY [1 .. MAX_VALUES] OF INTEGER** around. (Of course, any descendants of this object will also have this field — but, then these descendants are all **NrGrp**s.)
- I've introduced a shell — **PlainRand** — for the random number generator. We've been getting along fine with **RAND**, and it works quite nicely. However, sometimes it may be convenient to use a different generator — for example, if you need to generate special kinds of distributions, or if you plan to move your program to an environment that doesn't use **RAND**. (Other Pascal implementations, for example, use **RANDOMIZE** and **RANDOM** in place of **SRAND** and **RAND**, respectively.)

Numerical Examination ... This is Not a Test

Usually — at least outside the world of programming books — numbers are acquired or generated for a reason. Once you have the numbers, you'll want to know something about them. *Descriptive Statistics* provide summary information about a collection of numbers.

Values such as the *mean* (often referred to, mistakenly, as the average), the *median* (middle value, based on size), and *mode* (most frequently occurring value) all can give you some sense of the "typical" value in a collection.

The mean, median, or mode are, technically, all "averages," which is why it's not quite correct to use average to refer to the mean. But don't worry, everyone does it, and everyone will understand what you mean. Be aware, however, that there can be differences, and these differences can have consequences. For example, the mean income is almost always higher than the median income — because a small fraction of incomes is much higher than the rest, which pulls the mean toward the high end. The median, in contrast, is based on the number of people at each income level, and so is actually insensitive to actual income levels. Keep that in mind the next time someone or some report tells you the average income has risen. (Incidentally, if you're interested in such things — or even if you're not — I highly recommend a book called *How to Lie with Statistics* by Darrel Huff and Irving Geis. It will completely change the way you read the newspapers and watch TV.)

Averages try to capture a typical case. Other descriptive statistics try to provide information about the extent to which the numbers are spread out or scattered. Range and standard deviation are two examples of such statistics. The *range* represents the difference between the largest and the smallest values in a collection. (The range is sensitive to extreme values which are, by definition, rare; consequently, a somewhat more reliable range value is the *interquartile range*, which represents the distance between the bottom and top quarters of the collection.) The standard deviation provides a weighted measure of the spread of values in a collection. It has properties that make it a useful number when doing *inferential statis-*

tics (which try to enable you to draw conclusions from or about your numbers, and to assess the confidence you should have in these conclusions.)

This numerical homily is to explain why the descendant object we'll create in this section is a useful one. In this section, we'll develop an object class based on **NrGrp**. The **Stat** object adds some statistical (surprise, surprise!) routines to the **NrGrp** object. These methods enable you to explore the numbers in more detail. Here's the object's definition:

```
TYPE Stat = OBJECT ( NrGrp)
       Mean : REAL;
       Minimum, Maximum, Range : INTEGER;
       PROCEDURE Init; OVERRIDE;
       FUNCTION FindMean : REAL;
       FUNCTION FindMin : INTEGER;
       FUNCTION FindMax : INTEGER;
       FUNCTION FindRange : INTEGER;
     END;   (* Stat OBJECT *)
```

This object introduces the fields needed to store some statistical values, as well as the methods for computing and returning these values.

The methods are defined as follows:

```
PROCEDURE Stat.Init;
VAR Index : INTEGER;
BEGIN
  INHERITED Init;
  Mean := -32768;
  Maximum := -32768;
  Minimum := MAXINT;
END; (* Stat.Init *)

FUNCTION Stat.FindMin : INTEGER;
VAR Index : INTEGER;
BEGIN
  FOR Index := 1 TO NrVals DO
    IF Nrs [ Index] < Minimum THEN
       Minimum := Nrs [ Index];
  FindMin := Minimum;
END; (* FUNCTION Stat.FindMin *)

FUNCTION Stat.FindMax : INTEGER;
VAR Index : INTEGER;
```

```
BEGIN
  FOR Index := 1 TO NrVals DO
    IF Nrs [ Index] > Maximum THEN
       Maximum := Nrs [ Index];
  FindMax := Maximum;
END; (* FUNCTION Stat.FindMax *)

FUNCTION Stat.FindRange : INTEGER;
BEGIN
  Range := FindMax - FindMin;
  FindRange := Range;
END;  (* FUNCTION Stat.FindRange *)

FUNCTION Stat.FindMean : REAL;
CONST ERROR_VAL = -99999.99;
VAR Index : INTEGER;
    RunningSum : REAL;
BEGIN
  RunningSum := 0;
  FOR Index := 1 TO NrVals DO
    RunningSum := RunningSum + Nrs [ Index];
  IF NrVals > 0 THEN
    FindMean := RunningSum / NrVals
  ELSE
    FindMean := ERROR_VAL;
END; (* FUNCTION Stat.FindMean *)
```

The **Init** method, which overrides its ancestor's version (but does use it), assigns "nonsense" values. These should be recognized immediately as being suspect if they are returned. (If the minimum value in your number collection is **MAXINT**, then you have one unusual collection.

In **FindMean**, note that **RunningSum** is defined as a **REAL**, rather than as a **LONGINT**. One reason for this is to ensure that the division is real-number (and not integer) division.

These methods actually compute their statistics each time they are called. To understand why, note that — for purposes of your programs — the numbers in the array don't have mean, maximum, minimum, or range values. Actually, they *do* have mean, minimum, maximum, and range values, but they don't have true **Mean**, Minimum, **Maximum**, and **Range** values until the associated methods are called. Fortunately, the associated methods are the only ways to get to the summary values — so you can be assured that the values are computed when you ask for them. Such protection is another nice feature of object-oriented programming — at least when you don't try to cheat by trying to get values directly.

The following program (**12-2.PAS**) adds the **Stat** object to the previous program, and also lets you test the new methods.

Listing 12.2

```
(* Program to illustrate definition and use of a Stat object *)

PROGRAM StatDemo;
USES Windows;

CONST NR_SAMPLES = 100;
      MAX_VALUES = 500;
      PER_LINE = 10;
TYPE NrGrp = OBJECT
       NrVals : INTEGER;  (* # numbers in the array *)
       Nrs : ARRAY [ 1 .. MAX_VALUES] OF INTEGER; (* number
container *)
       (* initialize the array and the counter *)
       PROCEDURE Init;
       (*
          Returns the # of values in the array
          --- ie, the value of NrVals
       *)
       FUNCTION ShowSize : INTEGER;
       (* Write a value to the next available cell *)
       PROCEDURE PutVal ( TheVal : INTEGER);
       (* Retrieve a value from a specified cell in the array *)
       FUNCTION GetVal ( Index : INTEGER) : INTEGER;
       (* Search through the array cells for a specified value *)
       FUNCTION FindVal ( TheVal : INTEGER) : INTEGER;
     END;  (* NrGrp OBJECT *)

     Stat = OBJECT ( NrGrp)
       Mean : REAL;
       Minimum, Maximum, Range : INTEGER;
       PROCEDURE Init; OVERRIDE;
       FUNCTION FindMean : REAL;
       FUNCTION FindMin : INTEGER;
       FUNCTION FindMax : INTEGER;
       FUNCTION FindRange : INTEGER;
     END;  (* Stat OBJECT *)

VAR MyNrs : NrGrp;
```

```
    MyStats : Stat;
    Index, HowManyCells, RandSeed : INTEGER;

(* ************************************** *)
(*        Method Definitions              *)
(* ************************************** *)

   (* Initialize the object's data elements *)
   PROCEDURE NrGrp.Init;
   VAR Index : INTEGER;
   BEGIN
     FOR Index := 1 TO MAX_VALUES DO
       Nrs [ Index] := 0;
     NrVals := 0;
   END; (* NrGrp.Init *)

  (* Return the number of elements currently in the array *)
  FUNCTION NrGrp.ShowSize : INTEGER;
  BEGIN
    ShowSize := NrVals;
  END; (* NrGrp.ShowSize *)

  (* Add a value to the array *)
  PROCEDURE NrGrp.PutVal ( TheVal : INTEGER);
  BEGIN
    IF NrVals <= MAX_VALUES THEN
    BEGIN
      INC ( NrVals);
      Nrs [ NrVals] := TheVal;
    END;
  END;  (* NrGrp.PutVal *)

  (* Retrieve the contents of the specified cell *)
  FUNCTION NrGrp.GetVal ( Index : INTEGER) : INTEGER;
  CONST ERROR_VAL = -32768;
  BEGIN
    (* Nake sure the requested value is within the valid range *)
    IF ( Index >= 1) AND ( Index <= NrVals) THEN
      GetVal := Nrs [ Index]
    ELSE
    GetVal := ERROR_VAL;
END;  (* FUNCTION NrGrp.GetVal *)

(* Search the array for a specified data value *)
FUNCTION NrGrp.FindVal ( TheVal : INTEGER) : INTEGER;
VAR Index : INTEGER;
    Found : BOOLEAN;
BEGIN
  Found := FALSE;
  Index := 1;
  WHILE ( Index <= NrVals) AND ( NOT Found) DO
  BEGIN
    IF Nrs [ Index] = TheVal THEN
      Found := TRUE
    ELSE
      INC ( Index);
  END;
  IF Found THEN
    FindVal := Index
  ELSE
    FindVal := 0;
END; (* FUNCTION NrGrp.FindVal *)

PROCEDURE Stat.Init;
VAR Index : INTEGER;
BEGIN
  INHERITED Init;
  Mean := -32768;
  Maximum := -32768;
  Minimum := MAXINT;
END; (* Stat.Init *)

FUNCTION Stat.FindMin : INTEGER;
VAR Index : INTEGER;
BEGIN
  FOR Index := 1 TO NrVals DO
    IF Nrs [ Index] < Minimum THEN
       Minimum := Nrs [ Index];
  FindMin := Minimum;
END; (* FUNCTION Stat.FindMin *)
```

```
  FUNCTION Stat.FindMax : INTEGER;
  VAR Index : INTEGER;
  BEGIN
    FOR Index := 1 TO NrVals DO
      IF Nrs [ Index] > Maximum THEN
         Maximum := Nrs [ Index];
    FindMax := Maximum;
  END; (* FUNCTION Stat.FindMax *)

  FUNCTION Stat.FindRange : INTEGER;
  BEGIN
    Range := FindMax - FindMin;
    FindRange := Range;
  END;  (* FUNCTION Stat.FindRange *)

  FUNCTION Stat.FindMean : REAL;
  CONST ERROR_VAL = -99999.99;
  VAR Index : INTEGER;
      RunningSum : REAL;
  BEGIN
    RunningSum := 0;
    FOR Index := 1 TO NrVals DO
      RunningSum := RunningSum + Nrs [ Index];
    IF NrVals > 0 THEN
      FindMean := RunningSum / NrVals
    ELSE
      FindMean := ERROR_VAL;
  END; (* FUNCTION Stat.FindMean *)

(* ************************************ *)
(*         Auxiliary Routines           *)
(* ************************************ *)

  (* A shell within which you can put whatever random
     number generator you might want to use.
     In this version, it just calls the RAND function.
  *)
  FUNCTION PlainRand : INTEGER;
  BEGIN
```

```
    PlainRand := RAND;
  END;  (* FUNCTION PLainRand *)

  (* Get a number from the user *)
  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

(* ************************************* *)
(*      Program-Specific Routines        *)
(* ************************************* *)

  (* Examine the values of a sample of cells, and report
     some information about the properties of these values.
     NOTE: OddLead represents how often NrOdd was greater than
           NrEven when we checked; EvenLead represents the # of
           times NrEven was in the lead; Ties represents the # of
           times NrEvn and NrOdd were equal when we checked.
  *)
  PROCEDURE SampleCells;
  VAR Index,
      Result, WhichCell,
      NrOdd, NrEven,
      OddLead, EvenLead, Ties : INTEGER;
  BEGIN
    NrOdd := 0;  (* # of odd values in the sample *)
    NrEven := 0; (* # of even values in the sample *)

    OddLead := 0; EvenLead := 0; Ties := 0;
    FOR Index := 1 TO NR_SAMPLES DO
    BEGIN
      (* select a cell, to see its value *)
      WhichCell := PlainRand MOD MAX_VALUES;
      Result := MyNrs.GetVal ( WhichCell);
      IF ODD ( Result) THEN
        INC ( NrOdd)
      ELSE
        INC ( NrEven);
      IF NrOdd > NrEven THEN
        INC ( OddLead)
      ELSE IF NrEven > NrOdd THEN
        INC ( EvenLead)
      ELSE
        INC ( Ties);
    END;
    WRITELN ( 'Of ', NR_SAMPLES : 0, ' cells checked, ',
                     NrOdd : 0, ' had odd values, and ',
                     NrEven : 0, ' had even values.');
    WRITELN ( 'During the sampling, Odd was leading ', OddLead :
0,
              ' times, ');
    WRITELN ( 'even was leading ', EvenLead : 0,
              ' times, and there were ', Ties : 0, ' ties.');
  END; (* SampleCells *)

  PROCEDURE ShowNumbers (NrCells : INTEGER);
  VAR Index : INTEGER;
  BEGIN
    IF NrCells > MAX_VALUES THEN
    BEGIN
      WRITELN ( 'There are only ', MAX_VALUES:0,
                ' cells to see.');
      NrCells := MAX_VALUES;
    END;
    WRITELN ( 'The values in the first ', NrCells:0,
              ' cells are shown, 10 values per line.');
    FOR Index := 1 TO NrCells DO
    BEGIN
      WRITE ( MyNrs.GetVal ( Index) : 7);
      IF ( Index MOD PER_LINE) = 0  THEN
```

```
        WRITELN;
    END;
  END;  (* ShowNumbers *)

BEGIN  (* Main Program *)
  GetInt ( 'Seed for the Random Number Generator?', RandSeed);
  SRAND ( RandSeed);
  (* Allocate storage for the object --- Now a Stat object *)
  NEW ( MyStats);
  MyStats.Init;   (* Initialize it *)

  (* Feed some values to the number group array *)
  FOR Index := 1 TO MAX_VALUES DO
    MyStats.PutVal ( PlainRand);
{
  (* WARNING!!!!!
     If you decide to remove the comments around the code from
the
     previous program, MAKE SURE you change the calls in
SampleCells
     and in ShowNumbers --- from MyNrs to MyStats.
     If you don't, expect surprises.
  *)
  SampleCells;

  GetInt ( 'Display how many cell values?', HowManyCells);
  ShowNumbers ( HowManyCells);
}

  WRITELN ( MyStats.ShowSize : 0, ' Values range over ',
            MyStats.FindRange, ' points, ');
  WRITELN ( 'with a low of ', MyStats.FindMin,
            ', and a high of ', MyStats.FindMax);
  WRITELN ( 'Mean value is ', MyStats.FindMean : 10 : 3);

  AwaitUser;
END.
```

I've left in the calls to the routines from the **NrGrp** program in case you want to play with them as well in the program. However, they're commented out, so you don't call them without first making some necessary changes. Those routines call **MyNrs**, which is not used in this program. To use the **SampleCells** and the **ShowNumbers** routines, make sure to change the variable references to **MyStats**.

Possibilities

By adding a few very simple methods, and a couple of new number fields, we've managed to add considerable functionality to our number collection. If you want to explore this topic — and object programming — further, here are a few suggestions for directions you might take:

- Create objects to compute other statistics, such as the standard deviation, median, or interquartile range. Note that, for some of these values — in particular, median and interquartile range — you'll need to sort the data. Once you introduce a sorting method (for example, a version of bubblesort, which can sort values in place — ie, without requiring a separate data structure) you have created an object that you can use for all sorts of things — some of them completely unconnected to statistics or even to numbers. For example, you could use the array and the sorting routine to sort the letters in a word or sentence. To do this, you just need to feed the string's characters through the **ORD** function before putting them into the array, and through the **CHR** function before sending them back to the calling routines.
- You can create objects capable of doing simple graphing — for example, using text characters to plot frequencies. For this, you'll need to introduce methods for grouping values into intervals (since printing a 32,000+ range of values without grouping them into intervals is a singularly useless task). Once you have grouping capabilities, you can create an object for data gathering and organizing.

Taking an Object in a Different Direction

In deriving the **Stat** object from **NrGrp**, we were focusing on the data organization and description. To test the objects and methods, however, we relied on rand number generators to create the data.

What if we focused, instead, on the generation process, with the data description or application as secondary? the **ChanceEvent** object shown below is also a descendant of a **NrGrp**. As you'll see, however, this will take us in a very different direction as we derive objects based on **ChanceEvent**.

```
ChanceEvent = OBJECT (NrGrp)
  PossibleOutcomes : INTEGER;   (* # of possible outcomes *)
  PROCEDURE Init; OVERRIDE;
  PROCEDURE SetOutcomes ( NrOutcomes : INTEGER);
  FUNCTION GetNrOutcomes : INTEGER;
  FUNCTION GenerateOutCome : INTEGER;
END;  (* ChanceEvent OBJECT *)
```

This object introduces only one new field, but it's an important one. This is because the **Possible Outcomes** will determine either what range of values can be stored in the array or what values can be read out — depending on whether the chance events are generated and then stored in the array or whether the outcomes are based on the values in the array. In other words, if you toss a coin (a chance event that we'll objectify below), you can either check the outcome (heads or tails) and then store it (perhaps as 0 or 1, or as 1 or 2) in the array. Or, you can generate values and store them in the array, and then use these values to determine whether an outcome is a head or a tail (for example, an even or odd value).

The new methods are for checking and setting values. There is also one that is for generating an outcome. This function performs some sort of magic internally (nothing even remotely, Houdini-esque, however, as you'll see) and then comes back with one of the possible outcomes.

This sounds simple, and it is. However, once again, there is a lot behind this simplicity. Once you have that, you could use the method (or a suitable variant) to toss a coin, roll one die, roll multiple dice, deal cards, spin a roulette wheel, and even simulate activity on a waiting line in a bank or store.

Here are the method definitions for the **ChanceEvent** object:

```
  PROCEDURE ChanceEvent.Init;
  BEGIN
    INHERITED Init;
    PossibleOutcomes := 0;
  END; (* ChanceEvent.Init *)

  PROCEDURE ChanceEvent.SetOutcomes (NrOutcomes : INTEGER);
  BEGIN
    PossibleOutcomes := NrOutcomes;
  END; (* ChanceEvent.SetOutcomes *)

  FUNCTION ChanceEvent.GetNrOutcomes : INTEGER;
  BEGIN
  GetNrOutcomes := PossibleOutcomes;
END; (* FUNCTION ChanceEvent.GetNrOutcomes *)

FUNCTION ChanceEvent.GenerateOutcome : INTEGER;
BEGIN
  GenerateOutcome := (PlainRand MOD GetNrOutcomes) + 1;
END; (* FUNCTION ChanceEvent.GenerateOutcome *)
```

And program **ChanceEventDemo** lets you test the new object's methods.

Listing 12.3

```
(*
   Program to illustrate definition and use of a
   ChanceEvent object
*)

PROGRAM ChanceEventDemo;
USES Windows;

CONST NR_SAMPLES = 100;
      MAX_VALUES = 500;
      PER_LINE = 10;
      OUTCOME_VALS = 6;
TYPE NrGrp = OBJECT
       NrVals : INTEGER;  (* # numbers in the array *)
       Nrs : ARRAY [ 1 .. MAX_VALUES] OF INTEGER; (* number
container *)
       (* initialize the array and the counter *)
       PROCEDURE Init;
       (*
          Returns the # of values in the array
          --- ie, the value of NrVals
       *)
       FUNCTION ShowSize : INTEGER;
       (* Write a value to the next available cell *)
       PROCEDURE PutVal ( TheVal : INTEGER);
       (* Retrieve a value from a specified cell in the array *)
       FUNCTION GetVal ( Index : INTEGER) : INTEGER;
       (* Search through the array cells for a specified value *)
       FUNCTION FindVal ( TheVal : INTEGER) : INTEGER;
     END;  (* NrGrp OBJECT *)

     ChanceEvent = OBJECT (NrGrp)
       PossibleOutcomes : INTEGER;  (* # of possible outcomes *)
       PROCEDURE Init; OVERRIDE;
       PROCEDURE SetOutcomes ( NrOutcomes : INTEGER);
       FUNCTION GetNrOutcomes : INTEGER;
       FUNCTION GenerateOutCome : INTEGER;
     END;  (* ChanceEvent OBJECT *)

VAR MyNrs : NrGrp;
    MyEvent : ChanceEvent;
    NrOutcomes, Index, RandSeed : INTEGER;

  FUNCTION PlainRand : INTEGER; FORWARD;

(* ************************************* *)
(*       Method Definitions              *)
(* ************************************* *)

   (* Initialize the object's data elements *)
   PROCEDURE NrGrp.Init;
   VAR Index : INTEGER;
   BEGIN
     FOR Index := 1 TO MAX_VALUES DO
       Nrs [ Index] := 0;
     NrVals := 0;
   END; (* NrGrp.Init *)

  (* Return the number of elements currently in the array *)
  FUNCTION NrGrp.ShowSize : INTEGER;
  BEGIN
    ShowSize := NrVals;
  END; (* NrGrp.ShowSize *)

  (* Add a value to the array *)
```

```
PROCEDURE NrGrp.PutVal ( TheVal : INTEGER);
BEGIN
  IF NrVals <= MAX_VALUES THEN
  BEGIN
    INC ( NrVals);
    Nrs [ NrVals] := TheVal;
  END;
END;  (* NrGrp.PutVal *)

(* Retrieve the contents of the specified cell *)
FUNCTION NrGrp.GetVal ( Index : INTEGER) : INTEGER;
CONST ERROR_VAL = -32768;
BEGIN
  (* Nake sure the requested value is within the valid range *)
  IF ( Index >= 1) AND ( Index <= NrVals) THEN
    GetVal := Nrs [ Index]
  ELSE
    GetVal := ERROR_VAL;
END;  (* FUNCTION NrGrp.GetVal *)

(* Search the array for a specified data value *)
FUNCTION NrGrp.FindVal ( TheVal : INTEGER) : INTEGER;
VAR Index : INTEGER;
    Found : BOOLEAN;
BEGIN
  Found := FALSE;
  Index := 1;
  WHILE ( Index <= NrVals) AND ( NOT Found) DO
  BEGIN
    IF Nrs [ Index] = TheVal THEN
      Found := TRUE
    ELSE
      INC ( Index);
  END;
  IF Found THEN
    FindVal := Index
  ELSE
    FindVal := 0;
END; (* FUNCTION NrGrp.FindVal *)
```

```
  PROCEDURE ChanceEvent.Init;
  BEGIN
    INHERITED Init;
    PossibleOutcomes := 0;
  END; (* ChanceEvent.Init *)

  PROCEDURE ChanceEvent.SetOutcomes (NrOutcomes : INTEGER);
  BEGIN
    PossibleOutcomes := NrOutcomes;
  END; (* ChanceEvent.SetOutcomes *)

  FUNCTION ChanceEvent.GetNrOutcomes : INTEGER;
  BEGIN
    GetNrOutcomes := PossibleOutcomes;
  END; (* FUNCTION ChanceEvent.GetNrOutcomes *)

  FUNCTION ChanceEvent.GenerateOutcome : INTEGER;
  BEGIN
    GenerateOutcome := (PlainRand MOD GetNrOutcomes) + 1;
  END; (* FUNCTION ChanceEvent.GenerateOutcome *)

(* ************************************** *)
(*         Auxiliary Routines             *)
(* ************************************** *)

  (* A shell within which you can put whatever random
     number generator you might want to use.
     In this version, it just calls the RAND function.
  *)
  FUNCTION PlainRand : INTEGER;
  BEGIN
    PlainRand := RAND;
  END;  (* FUNCTION PLainRand *)

  (* Get a number from the user *)
  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

(* ************************************ *)
(*      Program-Specific Routines       *)
(* ************************************ *)

  PROCEDURE SummarizeOutcomes ( NrOutcomes : INTEGER);
  TYPE ValArray = ARRAY [ -1 .. 50] OF INTEGER;
  VAR Vals : ValArray;
      Index, Result : INTEGER;
  BEGIN
    FOR Index := -1 TO 50 DO
      Vals [ Index] := 0;
    FOR Index := 1 TO MAX_VALUES DO
    BEGIN
      Result := MyEvent.GetVal (Index);
      IF ( Result > NrOutcomes) OR (Result < 0) THEN
        INC ( Vals [ -1])
      ELSE
        INC ( Vals [ Result]);
    END;
    FOR Index := 1 TO NrOutcomes DO
       WRITELN ( 'Outcome ', Index : 0, ' occurred ',
                  Vals [ Index] : 5, ' times.');
    WRITELN ( 'Invalid outcomes occurred ', Vals [ -1] : 0,
              ' times');
  END;  (* SummarizeOutcomes *)

BEGIN  (* Main Program *)
  GetInt ( 'Seed for the Random Number Generator?', RandSeed);
  SRAND ( RandSeed);
  NEW ( MyNrs);
  NEW ( MyEvent);
  MyEvent.Init;
  (* Set possible outcomes for rolling a single die *)
  NrOutcomes := OUTCOME_VALS;
  MyEvent.SetOutcomes ( NrOutcomes);
  (* Fill the array with MAX_VALUES rolls of a die *)
  FOR Index := 1 TO MAX_VALUES DO
  BEGIN
    MyEvent.PutVal ( MyEvent.GenerateOutCome);
  END;
  SummarizeOutComes (NrOutcomes);

  AwaitUser;
END.
```

This program generates values for a chance event with six possible outcomes — such as when you roll a single die. When you run it, you'll find that each value comes up roughly the same number of times. (Actually, the random number generator isn't completely random. If you ever decide to turn these into more than just play programs, you'll probably want to find a more powerful generator.)

The **SummarizeOutcomes** routine does some minimal error checking. Again, to take these programs beyond the play and example level, you'll need to tighten up error-checking. It's probably a good idea to do it in the methods themselves. In this case, that would mean the **GenerateOutcome** method would need to make sure it sends back only valid outcomes.

Another issue to consider has to do with the **GenerateOutcome** function. In its current form, it makes an internal adjustment so as to return values in the range 1 .. **PossibleOutcomes**. You'll often see routines that return values in the 0 .. *Range - 1* range. These are equivalent, but it's crucial to remember which you're working with.

Alea Iacta Est

The interpretation of the outcomes in the previous examples suggests a direction in which we can take the new object. Since — as the title of this section says — "the die is cast," let's go one step further and cast it in code. In this section we'll create a **DieRoll** object, which we'll use as a starting point for some explorations.

The new object and its methods are as follows:

```
     DieRoll = OBJECT ( ChanceEvent)
       Freqs : ARRAY [ 0 .. 6] OF INTEGER;
       PROCEDURE Init; OVERRIDE;
       (* add 1 to the frequency for the value rolled *)
       PROCEDURE AddToCount ( WhichVal : INTEGER);
       (* find out how often WhichVal has come up *)
       FUNCTION GetCount ( WhichVal : INTEGER) : INTEGER;
       (* generate an outcome --- ie, roll a die *)
       FUNCTION Roll : INTEGER;
     END;

(* ******************************************* *)
(*              DieRoll Methods                *)
(* ******************************************* *)

  PROCEDURE DieRoll.Init;
  VAR Index : INTEGER;
  BEGIN
    INHERITED Init;
    SetOutcomes (6);
    FOR Index := 1 TO PossibleOutcomes DO
      Freqs [ Index] := 0;
  END; (* DieRoll.Init *)

  FUNCTION DieRoll.Roll : INTEGER;
  BEGIN
    Roll := INHERITED GenerateOutcome;
  END; (* FUNCTION DieRoll.Toss *)

  PROCEDURE DieRoll.AddToCount ( WhichVal : INTEGER);
  BEGIN
   IF (WhichVal >= 1) AND ( WhichVal <= PossibleOutcomes) THEN
      INC ( Freqs [ WhichVal])
    ELSE
      WRITELN ( 'Invalid request. No changes made.');
  END; (* DieRoll.AddToCount *)

  FUNCTION DieRoll.GetCount ( WhichVal : INTEGER) : INTEGER;
  CONST ERROR_VAL = -1;
  BEGIN
   IF (WhichVal >= 1) AND ( WhichVal <= PossibleOutcomes) THEN
      GetCount := Freqs [ WhichVal]
    ELSE
    BEGIN
      WRITELN ( 'Invalid request. Nothing to return.');
      GetCount := ERROR_VAL;
    END;
  END; (* FUNCTION DieRoll.GetCount *)
```

Program **DieRollDemo** is very similar to the previous one — except that it instantiates an object that is more familiar than **ChanceEvent**. Recasting an object in specific terms is a common practice, and is often a great convenience for both developer and user. Notice that the new object does add functionality, and it also adds fields that can provide information for the kinds of explorations we did earlier.

Listing 12.4

```
(*
   Program to illustrate definition and use of a
   DieRoll object
*)

PROGRAM DieRollDemo;
USES Windows;

CONST NR_SAMPLES = 100;
      MAX_VALUES = 500;
      PER_LINE = 10;
      OUTCOME_VALS = 6;
TYPE NrGrp = OBJECT
       NrVals : INTEGER;  (* # numbers in the array *)
       Nrs : ARRAY [ 1 .. MAX_VALUES] OF INTEGER; (* number
container *)
       (* initialize the array and the counter *)
       PROCEDURE Init;
       (*
          Returns the # of values in the array
          --- ie, the value of NrVals
       *)
       FUNCTION ShowSize : INTEGER;
       (* Write a value to the next available cell *)
       PROCEDURE PutVal ( TheVal : INTEGER);
       (* Retrieve a value from a specified cell in the array *)
       FUNCTION GetVal ( Index : INTEGER) : INTEGER;
       (* Search through the array cells for a specified value *)
       FUNCTION FindVal ( TheVal : INTEGER) : INTEGER;
```

```
     END;  (* NrGrp OBJECT *)

     ChanceEvent = OBJECT (NrGrp)
       PossibleOutcomes : INTEGER;  (* # of possible outcomes *)
       PROCEDURE Init; OVERRIDE;
       (* set the value of PossibleOutcomes *)
       PROCEDURE SetOutcomes ( NrOutcomes : INTEGER);
       (* Determine the value of PossibleOutcomes *)
       FUNCTION GetNrOutcomes : INTEGER;
       (* Generate one of the possible outcomes *)
       FUNCTION GenerateOutCome : INTEGER;
     END;  (* ChanceEvent OBJECT *)

     DieRoll = OBJECT ( ChanceEvent)
       Freqs : ARRAY [ 0 .. 6] OF INTEGER;
       PROCEDURE Init; OVERRIDE;
       (* add 1 to the frequency for the value rolled *)
       PROCEDURE AddToCount ( WhichVal : INTEGER);
       (* find out how often WhichVal has come up *)
       FUNCTION GetCount ( WhichVal : INTEGER) : INTEGER;
       (* generate an outcome --- ie, roll a die *)
       FUNCTION Roll : INTEGER;
     END;

VAR MyNrs : NrGrp;
    MyEvent, MyTest : ChanceEvent;
    MyRoll : DieRoll;
    NrOutcomes, Index, Result, RandSeed : INTEGER;

  FUNCTION PlainRand : INTEGER; FORWARD;

(* ************************************** *)
(*       Method Definitions               *)
(* ************************************** *)

   (* Initialize the object's data elements *)
   PROCEDURE NrGrp.Init;
   VAR Index : INTEGER;
   BEGIN
     FOR Index := 1 TO MAX_VALUES DO
```

```
      Nrs [ Index] := 0;
    NrVals := 0;
  END; (* NrGrp.Init *)

 (* Return the number of elements currently in the array *)
 FUNCTION NrGrp.ShowSize : INTEGER;
 BEGIN
   ShowSize := NrVals;
 END; (* NrGrp.ShowSize *)

 (* Add a value to the array *)
 PROCEDURE NrGrp.PutVal ( TheVal : INTEGER);
 BEGIN
   IF NrVals <= MAX_VALUES THEN
   BEGIN
     INC ( NrVals);
     Nrs [ NrVals] := TheVal;
   END;
 END;  (* NrGrp.PutVal *)

 (* Retrieve the contents of the specified cell *)
 FUNCTION NrGrp.GetVal ( Index : INTEGER) : INTEGER;
 CONST ERROR_VAL = -32768;
 BEGIN
   (* Nake sure the requested value is within the valid range *)
   IF ( Index >= 1) AND ( Index <= NrVals) THEN
     GetVal := Nrs [ Index]
   ELSE
     GetVal := ERROR_VAL;
 END;  (* FUNCTION NrGrp.GetVal *)

 (* Search the array for a specified data value *)
 FUNCTION NrGrp.FindVal ( TheVal : INTEGER) : INTEGER;
 VAR Index : INTEGER;
     Found : BOOLEAN;
 BEGIN
   Found := FALSE;
   Index := 1;
   WHILE ( Index <= NrVals) AND ( NOT Found) DO
   BEGIN
```

```
    IF Nrs [ Index] = TheVal THEN
      Found := TRUE
    ELSE
      INC ( Index);
  END;
  IF Found THEN
    FindVal := Index
  ELSE
    FindVal := 0;
END; (* FUNCTION NrGrp.FindVal *)

PROCEDURE ChanceEvent.Init;
BEGIN
  INHERITED Init;
  PossibleOutcomes := 0;
END; (* ChanceEvent.Init *)

PROCEDURE ChanceEvent.SetOutcomes (NrOutcomes : INTEGER);
BEGIN
  PossibleOutcomes := NrOutcomes;
END; (* ChanceEvent.SetOutcomes *)

FUNCTION ChanceEvent.GetNrOutcomes : INTEGER;
BEGIN
  GetNrOutcomes := PossibleOutcomes;
END; (* FUNCTION ChanceEvent.GetNrOutcomes *)

FUNCTION ChanceEvent.GenerateOutcome : INTEGER;
BEGIN
  GenerateOutcome := (PlainRand MOD GetNrOutcomes) + 1;
END; (* FUNCTION ChanceEvent.GenerateOutcome *)

PROCEDURE DieRoll.Init;
VAR Index : INTEGER;
BEGIN
  INHERITED Init;
  SetOutcomes (6);
  FOR Index := 1 TO PossibleOutcomes DO
    Freqs [ Index] := 0;
END; (* DieRoll.Init *)
```

```
  FUNCTION DieRoll.Roll : INTEGER;
  BEGIN
    Roll := INHERITED GenerateOutcome;
  END; (* FUNCTION DieRoll.Toss *)

  PROCEDURE DieRoll.AddToCount ( WhichVal : INTEGER);
  BEGIN
    IF (WhichVal >= 1) AND ( WhichVal <= PossibleOutcomes) THEN
      INC ( Freqs [ WhichVal])
    ELSE
      WRITELN ( 'Invalid request. No changes made.');
  END; (* DieRoll.AddToCount *)

  FUNCTION DieRoll.GetCount ( WhichVal : INTEGER) : INTEGER;
  CONST ERROR_VAL = -1;
  BEGIN
    IF (WhichVal >= 1) AND ( WhichVal <= PossibleOutcomes) THEN
      GetCount := Freqs [ WhichVal]
    ELSE
    BEGIN
      WRITELN ( 'Invalid request. Nothing to return.');
      GetCount := ERROR_VAL;
    END;
  END; (* FUNCTION DieRoll.GetCount *)

(* ************************************** *)
(*        Auxiliary Routines              *)
(* ************************************** *)

  (* A shell within which you can put whatever random
     number generator you might want to use.
     In this version, it just calls the RAND function.
  *)
  FUNCTION PlainRand : INTEGER;
  BEGIN
    PlainRand := RAND;
  END;  (* FUNCTION PLainRand *)

  (* Get a number from the user *)
  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

(* ************************************ *)
(*      Program-Specific Routines        *)
(* ************************************ *)

  PROCEDURE SummarizeOutcomes ( NrOutcomes : INTEGER);
  TYPE ValArray = ARRAY [ -1 .. 50] OF INTEGER;
  VAR Vals : ValArray;
      Index, Result : INTEGER;
  BEGIN
    FOR Index := -1 TO 50 DO
      Vals [ Index] := 0;
    FOR Index := 1 TO MAX_VALUES DO
    BEGIN
      Result := MyEvent.GetVal (Index);
      IF ( Result > NrOutcomes) OR (Result < 0) THEN
        INC ( Vals [ -1])
      ELSE
        INC ( Vals [ Result]);
    END;
    FOR Index := 1 TO NrOutcomes DO
       WRITELN ( 'Outcome ', Index : 0, ' occurred ',
                   Vals [ Index] : 5, ' times.');
    WRITELN ( 'Invalid outcomes occurred ', Vals [ -1] : 0,
              ' times');
  END;  (* SummarizeOutcomes *)
  PROCEDURE CompareRolls;
  TYPE OutComeArray = ARRAY [ 1 .. 12] OF INTEGER;
  VAR Index, Roll1, Roll2 : INTEGER;
      Simple, Careful : OutcomeArray;
  BEGIN
    FOR Index := 1 TO 12 DO
    BEGIN
      Simple [ Index] := 0;
      Careful [ Index] := 0;
    END;
    MyEvent.Init;
    MyEvent.SetOutcomes ( 11);
    FOR Index := 1 TO MAX_VALUES DO
       MyEvent.PutVal ( MyEvent.GenerateOutcome + 1);
    FOR Index := 1 TO MAX_VALUES DO
    BEGIN
      INC ( Simple [ MyEvent.GetVal ( Index)]);
    END;

    MyTest.Init;
    MyTest.SetOutcomes ( 6);
    FOR Index := 1 TO MAX_VALUES DO
    BEGIN
      Roll1 := MyTest.GenerateOutcome;
      Roll2 := MyTest.GenerateOutcome;
      MyTest.PutVal ( Roll1 + Roll2);
    END;
    FOR Index := 1 TO MAX_VALUES DO
    BEGIN
      INC ( Careful [ MyTest.GetVal ( Index)]);
    END;

    WRITELN ( '   Index  Simple  Careful');
    FOR Index := 2 TO 12 DO
      WRITELN ( Index, Simple [ Index], Careful [ Index]);

  END; (* CompareRolls *)

BEGIN  (* Main Program *)
  GetInt ( 'Seed for the Random Number Generator?', RandSeed);
  SRAND ( RandSeed);
  NEW ( MyEvent);
  NEW ( MyTest);
  NEW ( MyRoll);
  MyRoll.Init;
  NrOutcomes := OUTCOME_VALS;
{
  (* The following statement is unnecessary for a DieRoll *)
  MyRoll.SetOutcomes ( NrOutcomes);
}
  (* Fill the array with MAX_VALUES rolls of a die *)
  FOR Index := 1 TO MAX_VALUES DO
  BEGIN
    Result := MyRoll.Roll;
    MyRoll.PutVal ( Result);
    MyRoll.AddToCount ( Result);
  END;

  FOR Index := 1 TO NrOutcomes DO
    WRITELN ( 'Outcome ', Index : 0, ' occurred ',
              MyRoll.GetCount ( Index) : 5, ' times.');

  AwaitUser;
END.
```

Nonetheless, if you've looked carefully, you'll see that there is a procedure defined that is not called in the program. The **CompareRolls** procedure raises an issue that is important if you're doing simulations — namely, the best way to generate values for the simulation.

Suppose you wanted to simulate the roll of two dice, for which the possible outcomes would be values from 2 to 12. One approach would be simply to use 11 possible outcomes and then add 1. This is what **CompareRolls** does with **MyEvent**. In contrast, the the routine rolls a single die twice to generate the values in **MyTest**. (This is a variable I snuck into the program for just this program. Were you paying attention?)

Add a call to **CompareRolls** near the end of your program, then compile and run it. Notice that method produces a roughly even distribution of values over the range. In contrast, method two (rolling one die twice) produces a distribution that is fat in the middle and sparse at the extremes. Which is the better simulation?

As it turns out, the second is, because not all possible values are equally likely when you roll two dice. This is because there are multiple ways of getting certain outcomes. For example, there's only one way to roll a 2 — by rolling a 1 on each die. In contrast, you can get a 7 in several ways : 6+1, 5+2, 4+3, 3+4, 2+5, 1+6. So, a roll of 7 should be six times as likely as a roll of 2. The second method produces results closer to such a distribution.

I've also let the **Summarize Outcomes** routine in — in case you want to practice by adapting this routine to this program.

A Final Exploration

As a final example, let's explore a claim I made earlier, and see whether it holds for dice as well as for coin tosses. Earlier in the chapter, I claimed that even for equally probable outcomes, one outcome would be in the lead in a majority of trials — so that it's difficult to catch up and take the lead once you fall behind in a game of chance. Does this pattern hold true with a die roll?

You've seen that a chance event with six possible outcomes spreads its outcomes more or less evenly across the possible outcomes. Does the outcome lead — the most frequent outcome until that point — also change?

With a truly uniform distribution, you would expect each value to have the lead about 16-17 percent of the time, or perhaps a bit less given that there may be ties. Let's say about 14 percent of the time (roughly one in every seven trials). Over 500 total tries, each player should be in the lead about 70 times or so.

The following program, **Explorer**, lets you find out for yourself. As with our other programs, this one generates 500 values, and then reports some information about the outcomes.

Listing 12.5

```
(*
   Program to illustrate definition and use of a
   DieRoll object
*)

PROGRAM Explorer;
USES Windows;

CONST NR_SAMPLES = 100;
      MAX_VALUES = 500;
      PER_LINE = 10;
      OUTCOME_VALS = 6;
TYPE NrGrp = OBJECT
       NrVals : INTEGER;  (* # numbers in the array *)
       Nrs : ARRAY [ 1 .. MAX_VALUES] OF INTEGER; (* number
container *)
       (* initialize the array and the counter *)
       PROCEDURE Init;
       (*
          Returns the # of values in the array
          --- ie, the value of NrVals
       *)
       FUNCTION ShowSize : INTEGER;
       (* Write a value to the next available cell *)
       PROCEDURE PutVal ( TheVal : INTEGER);
       (* Retrieve a value from a specified cell in the array *)
       FUNCTION GetVal ( Index : INTEGER) : INTEGER;
       (* Search through the array cells for a specified value *)
       FUNCTION FindVal ( TheVal : INTEGER) : INTEGER;
     END;  (* NrGrp OBJECT *)

     ChanceEvent = OBJECT (NrGrp)
       PossibleOutcomes : INTEGER;  (* # of possible outcomes *)
       PROCEDURE Init; OVERRIDE;
       PROCEDURE SetOutcomes ( NrOutcomes : INTEGER);
       FUNCTION GetNrOutcomes : INTEGER;
       FUNCTION GenerateOutCome : INTEGER;
     END;  (* ChanceEvent OBJECT *)

     DieRoll = OBJECT ( ChanceEvent)
       Freqs : ARRAY [ 0 .. 6] OF INTEGER;
       PROCEDURE Init; OVERRIDE;
       (* add 1 to the frequency for the value rolled *)
       PROCEDURE AddToCount ( WhichVal : INTEGER);
       (* find out how often WhichVal has come up *)
       FUNCTION GetCount ( WhichVal : INTEGER) : INTEGER;
       (* generate an outcome --- ie, roll a die *)
       FUNCTION Roll : INTEGER;
     END;
     DieExplorer = OBJECT ( DieRoll)
       Leads : ARRAY [ 0 .. 6] OF INTEGER;
       PROCEDURE Init; OVERRIDE;
       (* Find the value that has occurred most often so far *)
       FUNCTION FindMaxFreq : INTEGER;
       (* Value X is leading,
          add 1 to the # of trials for which it has led *)
       PROCEDURE FindandUpdateLeader;
       (* show how often the specified value has been in the
lead *)
       FUNCTION ShowLead ( Which : INTEGER) : INTEGER;
       (* Tell the caller who has currently been in the lead
          on the most trials
       *)
       PROCEDURE ReportLeader;
     END;

VAR MyNrs : NrGrp;
    MyEvent, MyTest : ChanceEvent;
    MyRoll : DieRoll;
    MyExpl : DieExplorer;
    NrOutcomes, Index, Result, RandSeed : INTEGER;

  FUNCTION PlainRand : INTEGER; FORWARD;

(* ************************************* *)
(*       Method Definitions              *)
(* ************************************* *)

   (* Initialize the object's data elements *)
   PROCEDURE NrGrp.Init;
   VAR Index : INTEGER;
   BEGIN
     FOR Index := 1 TO MAX_VALUES DO
       Nrs [ Index] := 0;
     NrVals := 0;
   END; (* NrGrp.Init *)

  (* Return the number of elements currently in the array *)
  FUNCTION NrGrp.ShowSize : INTEGER;
```

```
BEGIN
  ShowSize := NrVals;
END; (* NrGrp.ShowSize *)

(* Add a value to the array *)
PROCEDURE NrGrp.PutVal ( TheVal : INTEGER);
BEGIN
  IF NrVals <= MAX_VALUES THEN
  BEGIN
    INC ( NrVals);
    Nrs [ NrVals] := TheVal;
  END;
END;  (* NrGrp.PutVal *)

(* Retrieve the contents of the specified cell *)
FUNCTION NrGrp.GetVal ( Index : INTEGER) : INTEGER;
CONST ERROR_VAL = -32768;
BEGIN
  (* Nake sure the requested value is within the valid range *)
  IF ( Index >= 1) AND ( Index <= NrVals) THEN
    GetVal := Nrs [ Index]
  ELSE
    GetVal := ERROR_VAL;
END;  (* FUNCTION NrGrp.GetVal *)

(* Search the array for a specified data value *)
FUNCTION NrGrp.FindVal ( TheVal : INTEGER) : INTEGER;
VAR Index : INTEGER;
    Found : BOOLEAN;
BEGIN
  Found := FALSE;
  Index := 1;
  WHILE ( Index <= NrVals) AND ( NOT Found) DO
  BEGIN
    IF Nrs [ Index] = TheVal THEN
      Found := TRUE
    ELSE
      INC ( Index);
  END;
  IF Found THEN
```

```
      FindVal := Index
    ELSE
      FindVal := 0;
  END; (* FUNCTION NrGrp.FindVal *)

  PROCEDURE ChanceEvent.Init;
  BEGIN
    INHERITED Init;
    PossibleOutcomes := 0;
  END; (* ChanceEvent.Init *)

  PROCEDURE ChanceEvent.SetOutcomes (NrOutcomes : INTEGER);
  BEGIN
    PossibleOutcomes := NrOutcomes;
  END; (* ChanceEvent.SetOutcomes *)

  FUNCTION ChanceEvent.GetNrOutcomes : INTEGER;
  BEGIN
    GetNrOutcomes := PossibleOutcomes;
  END; (* FUNCTION ChanceEvent.GetNrOutcomes *)

  FUNCTION ChanceEvent.GenerateOutcome : INTEGER;
  BEGIN
    GenerateOutcome := (PlainRand MOD GetNrOutcomes) + 1;
  END; (* FUNCTION ChanceEvent.GenerateOutcome *)

  PROCEDURE DieRoll.Init;
  VAR Index : INTEGER;
  BEGIN
    INHERITED Init;
    SetOutcomes (6);
    FOR Index := 1 TO PossibleOutcomes DO
      Freqs [ Index] := 0;
  END; (* DieRoll.Init *)

  FUNCTION DieRoll.Roll : INTEGER;
  BEGIN
    Roll := INHERITED GenerateOutcome;
  END; (* FUNCTION DieRoll.Toss *)
```

```
PROCEDURE DieRoll.AddToCount ( WhichVal : INTEGER);
BEGIN
  IF (WhichVal >= 1) AND ( WhichVal <= PossibleOutcomes) THEN
    INC ( Freqs [ WhichVal])
  ELSE
    WRITELN ( 'Invalid request. No changes made.');
END; (* DieRoll.AddToCount *)

FUNCTION DieRoll.GetCount ( WhichVal : INTEGER) : INTEGER;
CONST ERROR_VAL = -1;
BEGIN
  IF (WhichVal >= 1) AND ( WhichVal <= PossibleOutcomes) THEN
    GetCount := Freqs [ WhichVal]
  ELSE
  BEGIN
    WRITELN ( 'Invalid request. Nothing to return.');
    GetCount := ERROR_VAL;
  END;
END; (* FUNCTION DieRoll.GetCount *)

PROCEDURE DieExplorer.Init;
VAR Index : INTEGER;
BEGIN
  INHERITED Init;
  FOR Index := 0 TO 6 DO
    Leads [ Index] := 0;
END; (* DieExplorer.Init *)

FUNCTION DieExplorer.FindMaxFreq : INTEGER;
VAR CurrMax, CurrSecond,
    CurrLeader, CurrSecLeader, Index : INTEGER;
BEGIN
  CurrMax := -32767;
  CurrSecond := -32767;
  CurrLeader := -1;
  CurrSecLeader := -1;
   (* Find leader *)
  FOR Index := 1 TO NrOutComes DO
  BEGIN
    IF Freqs [ Index] > CurrMax THEN
```

```
      BEGIN
        CurrLeader := Index;
        CurrMax := Freqs [ Index];
      END;
    END;  (* for picking leader *)
    (* Find second highest --- to rule out ties *)
    FOR Index := 1 TO NrOutComes DO
    BEGIN
      (* compare everyone except the leader *)
      IF ( Freqs [ Index] > CurrSecond) AND ( Index <>
CurrLeader) THEN
      BEGIN
        CurrSecLeader := Index;
        CurrSecond := Freqs [ Index];
      END;
    END;  (* for picking second highest *)
    IF CurrMax >  CurrSecond THEN
      FindMaxFreq := CurrLeader
    ELSE
      FindMaxFreq := 0;
  END; (* FUNCTION DieExplorer.FindMaxFreq *)

  PROCEDURE DieExplorer.FindandUpdateLeader;
  BEGIN
    Result := FindMaxFreq;
    INC ( Leads [ Result]);
  END; (* DieExplorer.FindandUpdateLeader *)

  FUNCTION DieExplorer.ShowLead ( Which : INTEGER) : INTEGER;
  BEGIN
    ShowLead := Leads [ Which];
  END; (* DieExplorer.ShowLead *)

  PROCEDURE DieExplorer.ReportLeader;
  VAR Index, CurrLead, CurrMax : INTEGER;
  BEGIN
    CurrMax := -32767;
    CurrLead := -1;
    FOR Index := 0 TO PossibleOutcomes DO
    BEGIN
```

```
      IF Leads [ Index] > CurrMax THEN
      BEGIN
        CurrMax := Leads [ Index];
        CurrLead := Index;
      END;
    END;
    IF CurrLead = 0 THEN
      WRITELN ( 'Most often in the lead was TIES, with ',
                Leads [ 0]:0, ' leads.')
    ELSE
       WRITELN ( 'Most often in the lead was ', CurrLead : 0,
                 '  with ', Leads [ CurrLead]:0, ' leads.')
  END; (* DieExplorer.ReportLeader *)

(* ************************************** *)
(*         Auxiliary Routines             *)
(* ************************************** *)

  (* A shell within which you can put whatever random
     number generator you might want to use.
     In this version, it just calls the RAND function.
  *)
  FUNCTION PlainRand : INTEGER;
  BEGIN
    PlainRand := RAND;
  END;  (* FUNCTION PLainRand *)

  (* Get a number from the user *)
  PROCEDURE GetInt ( Message : STRING;
                     VAR Value : INTEGER);
  BEGIN
    WRITE ( Message, ' ');
    READLN ( Value);
  END; (* GetInt *)

  PROCEDURE AwaitUser;
  VAR Ch : CHAR;
  BEGIN
    WRITE ( 'Press ENTER to continue.');
    READLN ( Ch);
  END; (* AwaitUser *)

(* ************************************** *)
(*      Program-Specific Routines         *)
(* ************************************** *)

  PROCEDURE SummarizeOutcomes ( NrOutcomes : INTEGER);
  TYPE ValArray = ARRAY [ -1 .. 50] OF INTEGER;
  VAR Vals : ValArray;
      Index, Result : INTEGER;
  BEGIN
    FOR Index := -1 TO 50 DO
      Vals [ Index] := 0;
    FOR Index := 1 TO MAX_VALUES DO
    BEGIN
      Result := MyEvent.GetVal (Index);
      IF ( Result > NrOutcomes) OR (Result < 0) THEN
        INC ( Vals [ -1])
      ELSE
        INC ( Vals [ Result]);
    END;
    FOR Index := 1 TO NrOutcomes DO
       WRITELN ( 'Outcome ', Index : 0, ' occurred ',
                  Vals [ Index] : 5, ' times.');
    WRITELN ( 'Invalid outcomes occurred ', Vals [ -1] : 0,
              ' times');
  END;  (* SummarizeOutcomes *)

  PROCEDURE CompareRolls;
  TYPE OutComeArray = ARRAY [ 1 .. 12] OF INTEGER;
  VAR Index, Roll1, Roll2 : INTEGER;
      Simple, Careful : OutcomeArray;
  BEGIN
    FOR Index := 1 TO 12 DO
    BEGIN
      Simple [ Index] := 0;
      Careful [ Index] := 0;
    END;
    MyEvent.Init;
    MyEvent.SetOutcomes ( 11);
    FOR Index := 1 TO MAX_VALUES DO
       MyEvent.PutVal ( MyEvent.GenerateOutcome + 1);
    FOR Index := 1 TO MAX_VALUES DO
    BEGIN
      INC ( Simple [ MyEvent.GetVal ( Index)]);
    END;

    MyTest.Init;
    MyTest.SetOutcomes ( 6);
    FOR Index := 1 TO MAX_VALUES DO
    BEGIN
      Roll1 := MyTest.GenerateOutcome;
      Roll2 := MyTest.GenerateOutcome;
      MyTest.PutVal ( Roll1 + Roll2);
    END;
    FOR Index := 1 TO MAX_VALUES DO
    BEGIN
      INC ( Careful [ MyTest.GetVal ( Index)]);
    END;

    WRITELN ( '   Index  Simple  Careful');
    FOR Index := 2 TO 12 DO
      WRITELN ( Index, Simple [ Index], Careful [ Index]);

  END; (* CompareRolls *)

BEGIN  (* Main Program *)
  GetInt ( 'Seed for the Random Number Generator?', RandSeed);
  SRAND ( RandSeed);
  NEW ( MyEvent);
  NEW ( MyTest);
  NEW ( MyRoll);
  NEW ( MyExpl);
  MyExpl.Init;
  NrOutcomes := OUTCOME_VALS;
{
  (* The following statement is unnecessary for a DieExpl *)
  MyExpl.SetOutcomes ( NrOutcomes);
}
  (* Fill the array with MAX_VALUES rolls of a die *)
  FOR Index := 1 TO MAX_VALUES DO
  BEGIN
    Result := MyExpl.Roll;
    MyExpl.PutVal ( Result);
    MyExpl.AddToCount ( Result);
    MyExpl.FindAndUpdateLeader;
  END;

  FOR Index := 1 TO NrOutcomes DO
  BEGIN
    WRITE ( 'Outcome ', Index : 0, ' occurred ',
            MyExpl.GetCount ( Index) : 5, ' times, ');
    WRITELN ( 'and led on ', MyExpl.ShowLead ( Index) : 5,
                      ' rolls.');
  END;
  WRITELN;
  WRITELN ( ' Ties led on ', MyExpl.ShowLead ( 0) : 5,
            'rolls.');
  WRITELN;
  WRITELN ( 'From inside the Explorer: ');
  MyExpl.ReportLeader;

  AwaitUser;
END.
```

Objects for Play and for Serious Play

In this chapter, I've tried to present a range of things you can do starting from a common object. One purpose was to get you to see the considerable return you can get fro a minimal investment. If you look at the past few programs, you'll see object instances being called with methods from their distant ancestors, together with calls to newly defined methods.

Because of the way object technology and the **OVERRIDE** command, work you can do this without any difficulty. In fact, you don't even need access to the source code for ancestor objects. As long as the object is accessible — for example, through a unit interface — you can create descendant objects to suit your needs.

In this final section, I'll suggest some directions you can go with these objects as a starting point, and will mention a few of the considerations involved. If you're interested, you'll need to work out the details and do it.

Simulations

In a sense, rolling a die is already a simulation. However, you can get much more complicated by varying such things as the number and likelihood of outcomes, the number of times you fill the array — for example, to collect multiple samples.

Some of the more obvious simulations that come to mind immediately include:

- Coin tosses: collect repeated smaller samples of coin tosses, and count how often different numbers of heads come up within those samples. For example, if you collected a few hundred samples of 20 tosses each, and then examined how often 1, 2, 3, ... 18, 19, or 20 heads occur in each sample, you would create a *binomial* distribution (which summarizes the behavior of two possible outcomes over repeated trials, and which is used as the basis for several statistical tests). Generating repeated samples of a phenomenon, and then examining the behavior of variables over these samples is at the core or *Monte Carlo* simulation methods.

- Spinning roulette wheels and dealing card hands: just change the possible outcomes and generate values. For card decks, you would also need some way of mapping from outcome to a suit and face value combination, but this is certainly doable. (Remember from the two dice example, however, that you need to think carefully how to do this. For example, does it make more sense to generate a 13-possibility outcome (for face value) and then a 4-possibility outcome (for suit)? Or are you better off generating a 52-possibility outcome and then assigning the outcomes to cards on some predetermined basis?

- Biased outcomes: to create unfair coins or dice, you might just have to change the distribution of outcomes. For example, to simulate a coin biased toward landing heads up, you might want to make this outcome occur 60 percent of the time. To do that you would need to change the **GenerateOutcome** method. A simple solution is to increase the number of possible outcomes (let's say to 10), and then assign a proportionate number to heads and tails (for example, 6 for heads and 4 for tails). For more complex phenomena, you might need to use a different kind of generator — for example, one that uses a normal or a Poisson distribution. The random number generator behind **PlainRand** in the program is just the house **RAND** function, and this is designed to generate values according to a uniform distribution. A normal-distribution generator will produce values that fall into the bell-shaped distribution that has become so familiar in our culture. A Poisson distribution is one that generates data for events that happen at an average (excuse me, mean) rate, but that can vary with time. For example, you can use a Poisson distribution to simulate the arrivals and departures of customers on a bank or ticket line. Using such a generator, you can then compare the behavior of single-line setups (as you now see in most banks) as compared to multiple lines (as you still see in most supermarkets). Incidentally, I ran such simulations a few years ago (using a different, one-letter, language), and found that multiple line setups tended to have anywhere from 20 to 50 percent more people in them at any given time.

- Actual games: once you've developed methods for all the basic actions, you can easily provide the appropriate details for actual games. For example, once you've got a well-developed dice rolling mechanism, you just need to add a few methods to create a craps game or even a board game (although the latter has a lot of baggage related to the squares on the board an on any cards or other materials that need to be used. You may also need to keep track of the board.) As another example, creating a blackjack player is easy once you have the mechanisms for dealing a card or a hand.)

Summary

In this chapter we looked at some more issues related to object-oriented programming, and I tried to show you how fruitful this approach can be. In the next chapter, we're going to enter the world of Windows GUI programming — a world that's probably like nothing you've ever seen before. If you regard an object-oriented approach as the outer limits of programming, then surely Windows GUI programming is the Twilight Zone.

13

GUI Programming

... And what a sticky mess it can be. Welcome to the world of Windows programming!

In this chapter we can cover only a very few points about this task, which has filled probably several hundred books — most of them over 1000 pages and weighing 3 or 4 pounds. The really scary part is that most of those books actually do little more than summarize the procedures and functions available for programming in Windows. For example, I have, in front of me, four books about Windows programming. Two are function references, and two are programming guides. Together, they have over 4800 pages, and weigh 16 pounds.

Rather than try to make Windows programmers out of you, I'll just try to give you some glimpses into Windows programs and programming, and to provide some overview and some guidelines for helping you get oriented for Windows programming. I'll also show you some simple example programs, to give you some idea of what these programs look like.

Incidentally, I'm using Windows programming to refer to programming in the Windows 95 or Windows NT environments. While much of the discussion in this chapter might also apply to programming Windows 3.1, the details are almost certainly different, and many of the assumptions that hold for Windows 95 and NT programming do not hold for Windows 3.1. (Incidentally, a certain amount of the complexity of the Windows 95/NT programming environment stems from the efforts that Microsoft made to keep this programming environment back-compatible — as far as possible — with Windows 3.1.)

Windows Programs ...

Windows programs are event-driven. Put simply, as long as there is something to do, a Windows program continues executing. Of course, in such a case, executing may mean lying idle for a couple of weeks — until a scheduled or an unexpected event occurs.

Events may be initiated by a user, a device, another program, or even by Windows itself. Messages are used to communicate events to programs that are executing under Windows. At a given time, there may be half a dozen programs, or more, executing. Purists will argue that only one is *really* executing, and perhaps they're right.

But, regardless of what you call it, you can be working with any of half a dozen programs within a click or two of a mouse. For example, as I'm writing this, I have open the following projects: a DOS Prompt, a screen capture program, Adobe FrameMaker, Micrografx Picture Publisher, Microsoft Word, and The Metrowerks IDE. I can run any of the example programs — from this chapter or from earlier chapters, while all these other programs are accessible. Moreover, I can do this from inside the IDE or from the Windows desktop.

How is this possible? There are all sorts of reasons. Three of these are of particular interest for our purposes in this chapter:

- The whole process has a ringmaster: Windows 95 or Windows NT.
- Everything in a Windows program has a unique identity — at least in relation to a particular program or program element (such as a window).
- Windows program elements know how to take care of themselves, or they know how to call an element that does.

Because Windows program elements (windows, dialog boxes, etc.) are so active and self-managing, Windows can run things using messages. Each program has a *message queue*, which contains messages regarding events (actions to be taken or that have been taken by other programs, devices, or functions). Messages may tell the program to display something, to open, close, or move something, to quit, etc.

Messages have identities, destinations, and content. Each message has a unique identifier — at least for its destination. If a program has no messages in its queue, the program remains idle until it receives a message. If the message tells it to quit, the program terminates; if the message tells it to do something else, the program does it, and then checks whether there are any more messages.

In the DOS world, a running program basically took control of the system resources (memory, devices, etc.) and maintained control (or at least tried to) until it finished executing. Under certain circumstances, DOS programs could be stopped or preempted — by using low-level interrupts. But even that was not guaranteed to be successful, since some programs simply took control of the interrupts and refused to give them up.

Because they take control of the entire environment, DOS programs also become responsible for handling all devices and interfaces. In the DOS world, this means that you may need a different program to drive different monitors or different printers.

By contrast, in the Windows environment, Windows runs everything. In return, Windows takes care of handling interactions with devices and other programs. Because of this, Windows programs needn't worry about the details of how to display content for different devices or configurations.

Also, because Windows elements know how to carry out their tasks (move, open, close, etc.), Windows programs needn't be concerned with these details. Instead, Windows programs need to have provisions for handling the various events that are possible.

... And Windows Programming

Because of this, the main routine in Windows programs is often very simple — sometimes consisting of just a single call and a return value. The main function — known as **WinMain** — will generally call a Windows routine that is responsible for processing the event queue.

Windows programs rely extensively on the Windows API (Application Programming Interface). This is a collection of over 1000 procedures and functions that serve as a resource for programmers. Depending on the programming environment, programmers may access these routines directly, or they may call routines in some type of programming shell. For example, the Microsoft Foundation Class provides an object-oriented shell through which programs can call the API functions. In the example projects in this chapter, you'll be calling API functions directly.

Help for Windows Programs

To keep things more manageable, many Windows programs use special files to store resource information such as window and dialog box definitions, message content, etc. In fact, most Windows programs will have such resource files associated with it.

Let's try an example program, and look at the files associated with the project. The following listing shows the program in project 13-1.CWP.

Listing 13.1

```
(* Sample Windows-GUI Based Pascal Program  *)
(* CodeWarrior Pascal                        *)
(* © 1996 Metrowerks Corp.                   *)
(* For questions and comments:               *)
(*       support@metrowerks.com              *)
(*       http://www.metrowerks.com           *)

(* Delete this program from your project and    *)
(* add your own source file. It is included as  *)
(* a simple example and as a stationery         *)
(* placeholder only.                            *)

(* See the User's Guide for more information on *)
(* creating/manipulating projects and using     *)
(* Stationery.                                  *)
```

```
UNIT HelloWorld;

INTERFACE

USES
  Windows, resource;

FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;

IMPLEMENTATION

{$pragma noreturn on}

PROCEDURE HelloWorld_OnCommand ( _hwnd:HWND; id:LONGINT;
                                 hwndCtl:HWND;
                                 codeNotify:UINT); stdcall;
BEGIN
  CASE id OF
    IDCANCEL: EndDialog ( _hwnd, id);
  END;
END;


FUNCTION HelloWorld_DlgProc ( _hwnd:HWND; uMsg:UINT;
_wParam:WPARAM;
                              _lParam:LPARAM) : BOOL; STDCALL;
BEGIN
  HelloWorld_DlgProc := 0;
  CASE uMsg OF
    WM_COMMAND: BEGIN
      HelloWorld_OnCommand ( _hwnd, LONGINT(WORD(_wParam)),
                             HWND(_lParam),
                             UINT((_wParam SHR 16) AND
$0000FFFF));
      HelloWorld_DlgProc := 1;
    END
  END;
END;
```

```
FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;
BEGIN
  DialogBoxParamA ( hinstExe, LPCSTR(IDD_FORMVIEW),
                    NIL, HelloWorld_DlgProc, 0);
  WinMain := 0;
END;

{$pragma noreturn reset}


END.
```

The basic structure of this program consists of a call to **WinMain**, which calls an API routine named **DialogBoxParamA**. One of the parameters in this call is **HelloWorld_DlgProc**, which is a function written for this program. This function consists mainly of a CASE statement, with just a single case. For that case, **HelloWorld_DlgProc** calls **HelloWorld_OnCommand**, which is another routine written for this program.

As you examine the program, think about the program elements that you *don't* see. In particular, try to determine where the instructions are for displaying the window and its text.We'll address this issue in a bit.

There are several other things worth mentioning about this program:

- It's a Pascal unit. The Windows programs we'll be looking at in this chapter are all created as units. Note that only the **WinMain** function is defined in the interface. Note also that the interface **USES Windows** and another unit named **Resource**.
- The code seems to contain lots of different naming and capitalization styles. This actually turns out to be systematic (usually), and is done for a reason — to provide the programmer with useful information. The notation system is called Hungarian notation. and was developed by Charles Simonyi, one of Microsoft's ace programmers.
- **WinMain** is quite simple — consisting of a single call and a return value. Notice, however, that all these Windows-related routines have lots of parameters. That's something you'll have to get used to when creating Windows programs. One reason for this is because everything has an ID, which must be passed in function or procedure calls. So, when routines pass or process messages, or when they manipulate elements, then information about the elements involved must be passed.

- In Windows programs — especially now — you'll also see lots of numerical twisting and turning in parameters. One reason for this has to do with the back-compatibility issue. Windows 3.1 and its predecessors were 16-bit systems. From a programming standpoint, this means that addresses are stored in 16-bits. Windows 95 and Windows NT are 32-bit systems, which means four bytes are used for storing addresses. Many of the Windows routines are designed to be usable with either 16- or 32-bit platforms. Consequently, it's sometimes necessary to do type coercions — that is, to force parameters into the appropriate data size.

To build this project,

- Select **Open** from the **File** menu.
- Specify 13-1.cwp as the project name.
- Select **Make** from the **Project** menu.
- Select **Run** from the **Project** menu.

When you run the program, you'll see a Welcome message in a dialog box.

If you look at the listing for project 13-1, you won't see anything that even vaguely looks like either "Pascal Win32 Project Stationery" or "Welcome to CodeWarrior!" Where is this information, and how come it says Win32 Project Stationery?

Actually, what you've just done is compiled a version of the stationery for GUI projects. This program was written by Michael Cordery, one of the Metrowerks wizards who helped bring you this Pascal environment. We'll use this program as our template for the projects in this chapter.

Now, what about those strings displayed on the screen? Where do they come from, and how did tthey get put up on the screen? For the answer, look at the file **13-1.RC**, which should be located in your project folder, and which is shown in the following listing:

```
//Microsoft Developer Studio generated resource script.
//
#include "13-1.rh"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////
/////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
```

```
#define _WIN32
#include "windows.h"

//////////////////////////////////////////////////////////////
/////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////
/////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////
/////////////////
//
// Dialog
//

IDD_FORMVIEW DIALOG DISCARDABLE  50, 50, 117, 56
STYLE DS_MODALFRAME | DS_3DLOOK | DS_LEFT | WS_MINIMIZEBOX |
WS_VISIBLE |
    WS_CAPTION | WS_SYSMENU
CAPTION "Pascal Win32 Project Stationery"
FONT 14, "Arial"
BEGIN
    CTEXT           "Welcome to CodeWar-
rior!",IDC_STATIC,21,23,82,8,NOT
                    WS_GROUP
END


#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
/////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


////////////////////////////////////////////////////////////
/////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_FORMVIEW, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 110
        TOPMARGIN, 7
        BOTTOMMARGIN, 49
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////
/////////////////



#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
/////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////
/////////////////
#endif    // not APSTUDIO_INVOKED
```

This is an example of a *resource file*, and it contains the details behind the program's display. A resource file is used to store specific information for a project. The section that specifies the dialog box and its contents is the one under the Dialog heading. (Lines beginning with // in the RC file are comments).

The Dialog section does the following:

- Specifies a dialog box identified as **IDD_FORMVIEW**. The dialog box is positioned with its left corner at coordinate 50, 50; it is 117 units wide and 56 units high. (The coordinate system begins in the upper left corner of the main window, and is expressed in terms of the width of a character. Specifically, each horizontal unit is one fourth as wide as a character in the specified size and font; each vertical unit is one-eighth of the height of a character.) The **DISCARDABLE** tells Windows that the dialog box can be discarded from memory if that memory is needed for some other purpose.
- The **STYLE** line specifies some value that determine how the box will appear. For example, **DS_MODALFRAME** means that the box will become the active element when the program starts up; the **WS_MINIMIZEBOX** species that the box is to have a minimize box in the upper right area.
- The **FONT** line specifies what kind of font should be displayed, and at what size.
- Finally, the line between the **BEGIN** and **END**, contains the text that is displayed in the dialog box. The **CTEXT** says to center the string during output. The numbers near the end of the line specify a particular size and location.

Most of the file is administrative detail (which is essential for windows programs). The program-specific elements are found mainly in the DIALOG and the DESIGNINFO sections of the file. As you'll see, you can produce a different program simply by changing the contents of the RC file and then recompiling. (The RC file is actually processed by a Resource compiler program, which is part of the CodeWarrior environment.)

The text is actually displayed automatically, because the DIALOG was defined as active and visible (DS_MODALFRAME and WS_VISIBLE). So, simply by defining your program element the right way, you get automatic output. I told you these windows elements knew how to take care of themselves!

This program actually uses two additional files, which you can examine in your project folder. **RESOURCE.P** is a unit that contains some constant definitions, and **13-1.RH**, which contains definitions used by the RC file. (RH files sometimes have the extension H, instead of RH, as you'll see in some later examples.) Note that the example projects in this chapter all use their own **RESOURCE.P** files.

To see the advantage of such resource files, let's make a few changes in the resource file. Project 13-2 has the same structure as the stationery template, but includes a new resource file. The routines have different names, but are essentially the same as those in 13-1. Function **WinProc** has replaced **HelloWorld_DlgProc**, and the call to **DialogBoxParamA** has been replaced by a call to **DialogBox**.

Every windows program needs a **WinMain** function and a windows procedure to handle messages. This is usually called something like **WinProc** or **WndProc**. So, to make our examples look more like other windows programs you might study, the routine names have been changed. **DialogBox** is an API function that does the same sort of thing as **DialogBoxParamA**, but that has a simpler interface.

In the RC file for project 13-2, I just changed a few things in the dialog section: the dialog box dimensions, the string dimensions, and the string content. The listing for 13-2 shows the new program file. This is followed by the modified part of the RC file.

Listing 13.2

```
UNIT HelloWorld;

INTERFACE

USES
  Windows, resource;

FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;

IMPLEMENTATION

{$pragma noreturn on}

PROCEDURE HelloWorld_OnCommand ( _hwnd:HWND; id:LONGINT;
                                 hwndCtl:HWND;
                                 codeNotify:UINT); stdcall;
BEGIN
  CASE id OF
    IDCANCEL: EndDialog ( _hwnd, id);
  END;
END;


FUNCTION WinProc ( _hwnd:HWND; uMsg:UINT; _wParam:WPARAM;
                   _lParam:LPARAM) : BOOL; STDCALL;
```

```
BEGIN
  WinProc := 0;
  CASE uMsg OF
    WM_COMMAND: BEGIN
      HelloWorld_OnCommand ( _hwnd, LONGINT(WORD(_wParam)),
                             HWND(_lParam),
                             UINT((_wParam SHR 16) AND
$0000FFFF));
      WinProc := 1;
    END
  END;
END;


FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;
BEGIN
  DialogBox ( hinstExe, LPCSTR(IDD_FORMVIEW),
              NIL, WinProc);
  WinMain := 0;
END;

{$pragma noreturn reset}

END.
```

The following shows the section of the (13-2) RC file that differs from the first example. The only other difference is that 13-2.RC refers to 13-2.RH in its include statement.

```
/////
//
// Dialog
//

IDD_FORMVIEW DIALOG DISCARDABLE  50, 50, 167, 56
STYLE DS_MODALFRAME | DS_3DLOOK | DS_LEFT | WS_MINIMIZEBOX |
WS_VISIBLE |
    WS_CAPTION | WS_SYSMENU
CAPTION "My First Windows 95 Dialog"
FONT 14, "Arial"
BEGIN
```

```
    CTEXT            "Windows has a Lot of
Glass",IDC_STATIC,21,23,102,8,NOT
                     WS_GROUP
END
```

When you compile and run project 13-2, you'll see that it has different output. So, just by changing a few lines in a non-code file, I was able to create a new display. That's pretty neat, but it becomes really impressive when you think about all the functionality this little graphics element — the dialog box — already has. you can minimize it, move it (by dragging its title bar), and close it with a click of a mouse.

The windows routine (**WinProc**, in our example) calls routines for handling any of the messages it might receive. Since the example considers only one possible message, only one such routine (**HelloWorld_OnCommand**) is needed.

Project 13-3 shows how to define and use another windows element: a button. After displaying its dialog box (with some new text), the program also displays a button labeled "OK." Click on this to see what happens. Then check the routine for handling this message. That is, look at the routine (**ButtonExample_OnCommand**) called by WinProc. Notice that the "button" routine can deal with either of two cases: if you close the dialog box and if you click on the OK button.

Study the DIALOG section, to see what elements have been changed.

Listing 13.3

```
UNIT ButtonExample;

INTERFACE

USES
  Windows, resource;

FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;

IMPLEMENTATION

{$pragma noreturn on}

PROCEDURE ButtonExample_OnCommand ( _hwnd:HWND; id:LONGINT;
                                    hwndCtl:HWND;
```

```
                                             codeNotify:UINT); stdcall;
VAR
  szText, szCaption : PACKED ARRAY[0..127] OF CHAR;
BEGIN
  CASE id OF
    IDCANCEL: EndDialog(_hwnd, id);
    IDOK: BEGIN
      szCaption := "SiLLyBaLLS???";
      szText := "BOO!    ";
      MessageBox ( _hwnd, @szText, @szCaption,
                   UINT(MB_OK OR MB_ICONEXCLAMATION OR
MB_APPLMODAL));
    END
  END;
END;


FUNCTION WinProc ( _hwnd:HWND; uMsg:UINT; _wParam:WPARAM;
                   _lParam:LPARAM) : BOOL; STDCALL;
BEGIN
  WinProc := 0;
  CASE uMsg OF
    WM_COMMAND: BEGIN
      ButtonExample_OnCommand ( _hwnd, LONGINT(WORD(_wParam)),
                                HWND(_lParam),
                                UINT((_wParam SHR 16) AND
$0000FFFF));
      WinProc := 1;
    END
  END;
END;


FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;
BEGIN
  DialogBox ( hinstExe, LPCSTR(IDD_DIALOG1),
              NIL, WinProc);
  WinMain := 0;
```

```
END;

{$pragma noreturn reset}

END.
```

The contents of the RC file are shown in the following listing:

```
//Microsoft Developer Studio generated resource script.
//
#include "13-3.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////
/////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////
/////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////
/////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////
/////////////////
//
// Dialog
//

IDD_DIALOG1 DIALOG DISCARDABLE  0, 0, 186, 95
STYLE DS_MODALFRAME | DS_3DLOOK | WS_MINIMIZEBOX | WS_POPUP |
WS_CAPTION |
    WS_SYSMENU
CAPTION "My First Windows 95 Dialog"
FONT 10, "Arial"
```

```
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,67,58,50,14
    CTEXT          "Windows has a Lot of Glass, if you push
the right buttons",
                    IDC_STATIC,32,23,122,20
END


////////////////////////////////////////////////////////////
/////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 179
        TOPMARGIN, 7
        BOTTOMMARGIN, 88
    END
END
#endif    // APSTUDIO_INVOKED


#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
/////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "13-3.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END
```

```
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////
/////////////////



#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
/////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////
/////////////////
#endif    // not APSTUDIO_INVOKED
```

Build and run the program. See how many differences you notice between this output and the one from project 13-2. For example, did you notice the addition of the maximize box?

Again the changes are mainly in the dialog box and in one routine:

- Adding the **WS_MAXIMIZEBOX** to the style line gave us a working maximize box.
- Although **CTEXT** will automatically be wrapped within the dialog box, you won't see the material on the second line if you just add to the string. That's because the last number after the string specifies how deep the area for displaying text should be. This is 8 units in the original file — just enough room for one line of text. To ensure that any text that wraps around is also displayed, I made the text area deeper.

Adding some text strings and a call to **MessageBox** in **ButtonExample_OnCommand** produces the output when you click on the OK button.

The Program

Let's review: To follow program execution, go to the end of the source file for any of the first three projects, and start with the definition of **WinMain** at the end of the file.

This calls **DialogBox**, which is a predefined Windows function. Its job is to create a modal dialog box — that is, a dialog box that is active when it starts up. This function takes the following parameters:

- **hinstExe**, which is a handle for the program that is using the dialog box. (A *handle* is just a unique number associated with an object, and used to distinguish the object from other, similar, objects.
- The second parameter specifies the ID of string that contains the beginning of the dialog template. Note the **IDD_FORMVIEW** in this argument. That's the identifier associated with the Dialog in the resource file. (Incidentally, identifiers in all caps are used to indicate variables that have been defined in an include file or that represent a macro. This is one of the rules of Hungarian notation.)
- The third parameter represents the handle of the parent window. In Widows 95 and NT, this is always 0.
- The fourth parameter is a pointer to the routine that is handling messages for the program. In example 13-1, this was **HelloWorld_DlgProc**, and in the other examples it is **WinProc**.

More Windows Elements

Project 13-4 shows you how to define radio buttons.

Listing 13.4

```
UNIT RadioExample;

INTERFACE

USES
  Windows, resource;

FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;

IMPLEMENTATION
```

```
{$pragma noreturn on}

PROCEDURE RadioExample_OnCommand ( _hwnd:HWND; id:LONGINT;
                                   hwndCtl:HWND;
codeNotify:UINT); stdcall;
BEGIN
  CASE id OF
    IDCANCEL: EndDialog ( _hwnd, id);
  END;
END;


FUNCTION WinProc ( _hwnd:HWND; uMsg:UINT; _wParam:WPARAM;
                   _lParam:LPARAM) : BOOL; STDCALL;
BEGIN
  WinProc := 0;
  CASE uMsg OF
    WM_COMMAND: BEGIN
      RadioExample_OnCommand ( _hwnd, LONGINT(WORD(_wParam)),
                               HWND(_lParam),
                               UINT((_wParam SHR 16) AND
$0000FFFF));
      WinProc := 1;
    END
  END;
END;


FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;
BEGIN
  DialogBox ( hinstExe, LPCSTR(IDD_DIALOG1),
              NIL, WinProc);
  WinMain := 0;
END;

{$pragma noreturn reset}
```

```
END.
```

And its RC file is shown in the following:

```
//Microsoft Developer Studio generated resource script.
//
#include "13-4.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////
/////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////
/////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////
/////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////
/////////////////
//
// Dialog
//

IDD_DIALOG1 DIALOG DISCARDABLE  0, 0, 186, 95
STYLE DS_MODALFRAME | WS_MINIMIZEBOX | WS_POPUP | WS_CAPTION
| WS_SYSMENU
CAPTION "My Second (or so) Windows 95 Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL         "AM",IDC_RADIO1,"But-
ton",BS_AUTORADIOBUTTON,34,42,27,10
    CONTROL         "FM",IDC_RADIO2,"But-
ton",BS_AUTORADIOBUTTON,124,42,26,10
END
```

```
/////////////////////////////////////////////////////////////
/////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 179
        TOPMARGIN, 7
        BOTTOMMARGIN, 88
    END
END
#endif    // APSTUDIO_INVOKED


#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////
/////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "13-4.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////
/////////////////
```

```
#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
/////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////
/////////////////
#endif    // not APSTUDIO_INVOKED
```

The radio button element is defined within a dialog in the RC file.

The parameters for **RADIOBUTTON** are:

- Text that is to be displayed next to the button. This text must be within double quotes. If the text includes a quote, this must be replaced by two quotes.
- An ID for the button. If a dialog box has more than one radio button, each must have its own unique ID.
- The x-coordinate of the upper left corner of the button. This is in dialog units, and is relative to the upper left corner of the dialog box in which the radio button is found.
- The y-coordinate of the upper left corner of the button. This is in dialog units, and is relative to the upper left corner of the dialog box in which the radio button is found.
- The width of the button.
- The height of the button.
- An optional parameter lets you specify the style used for the button.

If you check the 13-4.H and the Resource.P files for this project,m you'll see definitions that are used for thias project. Recall that everything in a windows program needs a unique identifier. Header and resource files are where such identifiers are usually placed --- to get them out of the main program code. This makes the code cleaner, and also makes it easier to make changes for different environments. For example, if you wanted to run these programs in France, you could change the text to French in the RC file, rebuild the program. Presto! You now have a French dialog.

Project 13-5 shows you how to define an **EDITTEXT**, which is a control (window element) in which the user can edit text.

Listing 13.5

```
UNIT EditBoxExample;

INTERFACE

USES
  Windows, resource;

FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;

IMPLEMENTATION

{$pragma noreturn on}

PROCEDURE EditBoxExample_OnCommand ( _hwnd:HWND; id:LONGINT;
                                     hwndCtl:HWND;
codeNotify:UINT); stdcall;
BEGIN
  CASE id OF
    IDCANCEL: EndDialog ( _hwnd, id);
  END;
END;


FUNCTION WinProc ( _hwnd:HWND; uMsg:UINT; _wParam:WPARAM;
                   _lParam:LPARAM) : BOOL; STDCALL;
BEGIN
  WinProc := 0;
  CASE uMsg OF
    WM_COMMAND: BEGIN
      EditBoxExample_OnCommand ( _hwnd, LONGINT(WORD(_wParam)),
                                 HWND(_lParam),
                                 UINT((_wParam SHR 16) AND
$0000FFFF));
      WinProc := 1;
    END
  END;
END;


FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;
BEGIN
  DialogBox ( hinstExe, LPCSTR(IDD_DIALOG1),
              NIL, WinProc);
  WinMain := 0;
END;

{$pragma noreturn reset}

END.
```

Again, the contents of the RC file are shown in the following:

```
//Microsoft Developer Studio generated resource script.
//
#include "13-5.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////
/////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////
/////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////
/////////////////
// English (U.S.) resources
```

```
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

/////////////////////////////////////////////////////////////
/////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 179
        TOPMARGIN, 7
        BOTTOMMARGIN, 88
    END
END
#endif    // APSTUDIO_INVOKED


#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////
/////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "13-5.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END
```

```
#endif    // APSTUDIO_INVOKED


////////////////////////////////////////////////////////////
/////////////////
//
// Dialog
//

IDD_DIALOG1 DIALOGEX 0, 0, 186, 95
STYLE DS_MODALFRAME | DS_3DLOOK | WS_MINIMIZEBOX | WS_POPUP |
WS_CAPTION |
    WS_SYSMENU
CAPTION "My Third Windows 95 Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        IDC_EDIT1,28,18,130,58,ES_MULTILINE |
WS_VSCROLL,
                    WS_EX_CLIENTEDGE
END

#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////
/////////////////



#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
/////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////
/////////////////
#endif    // not APSTUDIO_INVOKED
```

Notice that the **EditBoxExample_OnCommand** routine is, again, a very simple one.

Finally, Project 13-6 shows you how to define a menu in the resource file. Note that certain menu items are designated as **POPUP**. These items display submenus when you select them. Notice that the POPUP items don't have their own identifiers. That's because you can't ever select a popup item. You can only select one of their subitems.

Listing 13.6

```
UNIT MenuExample;

INTERFACE

USES
  Windows, resource;

FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;

IMPLEMENTATION

{$pragma noreturn on}

PROCEDURE MenuExample_OnCommand ( _hwnd:HWND; id:LONGINT;
                                  hwndCtl:HWND;
codeNotify:UINT); stdcall;
BEGIN
  CASE id OF
    IDCANCEL: EndDialog ( _hwnd, id);
  END;
END;


FUNCTION WinProc ( _hwnd:HWND; uMsg:UINT; _wParam:WPARAM;
                   _lParam:LPARAM) : BOOL; STDCALL;
BEGIN
  WinProc := 0;
  CASE uMsg OF
    WM_COMMAND: BEGIN
      MenuExample_OnCommand ( _hwnd, LONGINT(WORD(_wParam)),
                              HWND(_lParam),
                              UINT((_wParam SHR 16) AND
$0000FFFF));
      WinProc := 1;
    END
  END;
```

```
END;


FUNCTION WinMain ( hinstExe:HINSTANCE; hinstPrev:HINSTANCE;
                   lpszCmdLine:LPSTR;
                   nCmdShow:LONGINT) : LONGINT; STDCALL;
BEGIN
  DialogBox ( hinstExe, LPCSTR(IDD_DIALOG1),
              NIL, WinProc);
  WinMain := 0;
END;

{$pragma noreturn reset}

END.
```

The RC file for this project provides you with a template you can use to create menus of your own.Notice that you can specify which character is to be underlined and used for selection from the menu.

```
//Microsoft Developer Studio generated resource script.
//
#include "13-6.h"

#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////
////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////
/////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////
/////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
```

```
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////
/////////////////
//
// Dialog
//

IDD_DIALOG1 DIALOG DISCARDABLE  0, 0, 186, 95
STYLE DS_MODALFRAME | WS_MINIMIZEBOX | WS_POPUP | WS_CAPTION
| WS_SYSMENU
CAPTION "My Fifth Windows 95 Dialog"
MENU IDR_MENU1
FONT 8, "MS Sans Serif"
BEGIN
END


////////////////////////////////////////////////////////////
/////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 179
        TOPMARGIN, 7
        BOTTOMMARGIN, 88
    END
END
#endif    // APSTUDIO_INVOKED


////////////////////////////////////////////////////////////
/////////////////
//
// Menu
//

IDR_MENU1 MENU DISCARDABLE
BEGIN
    POPUP "&Games"
    BEGIN
        MENUITEM "&Tag",                        ID_GAMES_TAG
      MENUITEM "&Hopscotch",              ID_GAMES_HOPSCOTCH
```

```
        POPUP "&Ring around the"
        BEGIN
            MENUITEM "R&osie",
ID_GAMES_RINGAROUNDTHE_ROSIE

            MENUITEM "&Mulberry Bush",
ID_GAMES_RINGAROUNDTHE_MULBERRYBUSH

            POPUP "&Sun"
            BEGIN
                MENUITEM "&Moon",
ID_GAMES_RINGAROUNDTHE_SUN_MOON

                MENUITEM "an&d",
ID_GAMES_RINGAROUNDTHE_SUN_AND

                MENUITEM "S&tars",
ID_GAMES_RINGAROUNDTHE_SUN_STARS

            END
        END
        MENUITEM "Musical &Chairs",
ID_GAMES_MUSICALCHAIRS
    END
END


#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
/////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "13-6.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END
```

```
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////
/////////////////



#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
/////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////
/////////////////
#endif    // not APSTUDIO_INVOKED
```

There is much more to programming the GUI than we've covered here. Because it would have been impossible to do anything involved, I opted for giving you an impressionistic introduction, along with some examples you can build upon to create your own programs. By helping you become familiar with some of the files and elements that are used in Windows programming, I hope the chapter has given you something to work with.

Happy GUI programming!

Index

A

ABS. See Function, ABS
Additive operator. See Operator, Additive
Address operator. See Operator, Address
Aggregate type. See Type, Aggregate
Alias 472
Ancestor object 559
APPEND procedure. See Procedure, APPEND
ARCTAN. See Function, ARCTAN
Argument 34, 156
Array cell 315
  Accessing 318–320
  Index versus Value 319
  Value versus index 319
Array index 315
ARRAY type. See Type, ARRAY
ASSIGN procedure. See Procedure, ASSIGN
Assignment operator. See Operator, Assignment

B

Binary file 396
Binding 585
  Early 585
  Late 586
Bit handling 528–551
Bitwise addition operator. See Operator, Bitwise addition
Bitwise AND operator. See Operator, Bitwise AND
Bitwise exclusion operator. See Operator, Bitwise exclusion
Bitwise multiplication operator. See Operator, Bitwise multiplication
Bitwise negation operator. See Operator, Bitwise negation
Bitwise NOT operator. See Operator, Bitwise NOT
Bitwise OR operator. See Operator, Bitwise OR
BlockRead procedure. See Procedure, BlockRead
BlockWrite procedure. See Procedure, BlockWrite
Boolean expression 130
BOOLEAN type. See Data type, BOOLEAN
Bubblesort Algorithm 64

C

CASE Statement. See Statement, CASE
Cell
  Accessing in a multidimensional array 330
  Index 315
Cell (array element) 315
CHAR type. See Data type, CHAR
Character-based type 122–125
CHR. See Function, CHR
Class 559
CLOSE procedure. See Procedure, CLOSE
CodeWarrior Pascal 17
  Naming rules 74
  Reserved keywords 76
  Runtime library 517
  Unit 517, 521, 521–528
  See also Pascal
Command Shortcuts 27
Commands
  Edit/Project Settings 20
  File/Open File 41
  Project/Make 39
Comments 72
  Nested 73
Comparison Operator. See Operator, Comparison
Compiler directive 283
Component array
  Accessing 330, 333
Compound Statement. See Statement, Compound
CONCAT. See Function, CONCAT
Concatenation operator. See Operator, Concatenation
Constant
  Definition 83
Constant Expression 85
Constants.P 521
Continuation condition 257
COPY. See Function, COPY
COS. See Function, COS
CYCLE. See Procedure, CYCLE

D

Data segment 461
Data type
  ARRAY
    Multidimensional 329–336
  ARRAY. See also Component array, Accessing
  BOOLEAN 128–140
  CHAR 122–125
  Character-based 122–125
  Creating 142
  DOUBLE 105
  Enumerated 141
  INTEGER 106–121
  Logical 128–140
  LONGINT 120
  LONGREAL 105
  OBJECT 559
    Representing 561
  Pointer 463–483
    Defining 465
    Pitfalls 477–483
    versus target type 470
  REAL 90–105
  Real number 90–105
  RECORD 341–364
    Variant 359–364
  SET 366–393
    Restrictions on 369
  SINGLE 105
  STRING 125–128, 325–329
    ARRAY of 334–336
  Subrange 339–341
  TEXT 396
  UNSIGNEDLONG 121
  UNSIGNEDWORD 121
  Whole number 106–121
Deallocation 479
DEC. See Procedure, DEC
Declaration
  FORWARD 275
DELETE. See Procedure, DELETE
Descendant object 559
Descriptive statistics 595
Dialog Box
  Project Settings 20
Dialog box
  New Project 24
Disjoint sets 382
DISPOSE procedure. See Procedure, DISPOSE
DOUBLE type. See Data type, DOUBLE
Dynamic memory allocation 462–463, 466–467

E

Early binding 585
Edit window 46
Empty set 370, 377, 383
Enumerated type. See Data type, Enumerated
Environment
  Immediate 279, 288, 299
  Local 279, 288, 299
EOF function. See Function, EOF
EOLN function. See Function, EOLN
Equality operator. See Operator, Equality
EXIT. See Procedure, EXIT
EXP. See Function, EXP
Exponentiation operator. See Operator, Exponentiation

F

Fibonacci number 305
Field
  Tag 362
Field (record) 341
  Structured type as 344
Field width
  for type REAL 94
File 395
  Adding to 422–425
  as parameter 408
  Binary 396
  Creating 398, 406
  Opening 408
  Text 396
  Typed 426–450
    Defining 430
  Untyped 454–459
File buffer 401
FOR Statement. See Statement, FOR
FORWARD Declaration 275
Function 165

- ABS 171
- Accessibility 286–300
- ARCTAN 172
- Body 167
- Call 167
- Calling 275–277
- CHR 173
- CONCAT 194
- COPY 196
- COS 175
- Definition 166
- EOF 412, 426
- EOLN 412, 426
- EXP 177
- IOResult 404, 426
- LENGTH 205
- LN 178
- MEMAVAIL 463
- Nested 277–279
- ODD 179
- ORD 181
- Parameter 170
- POS 206
- PRED 182
- RAND 208
- ROUND 184
- SIN 184
- SIZEOF 208
- SQR 187
- SQRT 189
- SUCC 191
- TRUNC 192
- UPCASE 211

Function. See also Scope
Functioncall
  Flow 303
Functions.P 522

G

Global routine 296
Greater than operator. See Operator, Greater than
Greater than or equal to operator. See Operator, Greater than or equal to

H

HALT. See Procedure, HALT
Heap 462

I

IDE 18, 29–31
  Edit window 46
  Starting 19, 29
Identifier 74
Identifier visbility. See Visibility
Identifiers
  Predefined 76
IF Statement. See Statement, IF
IF-ELSE Statement. See Statement, IF-ELSE
Immediate environment. See Environment, Immediate
IN operator. See Operator, IN
INC. See Procedure, INC
Include file 285
Index 315
Inequality operator. See Operator, Inequality
Inheritance 560
  Representing 563
INSERT. See Procedure, INSERT
INTEGER type. See Data type, INTEGER
Intersection operator. See Operator, Set intersection
IOResult function. See Function, IOResult
Iteration
  As alternative to recursion 311

K

Keywords
  Reserved 76

L

Late binding 586
LEAVE. See Procedure, LEAVE
Left shift operator. See Operator, Left shift
LENGTH. See Function, LENGTH
Less than operator. See Operator, Less than
Less than or equal to operator. See Operator, Less than or equal to
Linked list 484
LN. See Function, LN
Local environment. See Environment, Local

Local routine 296
Logical data type. See Data type, Logical
LONGINT type. See Data type, LONGINT
LONGREAL type. See Data type, LONGREAL
Loop variable. See Variable, Loop
Loop, FOR. See Statement, FOR

M

Macros.P 521
Mask 535
MEMAVAIL function. See Function, MEMAVAIL
Memory allocation
  Dynamic 462–463, 466–467
  Static 461–462
Memory deallocation 479
Method 568
  Overriding 574
Multidimensional array. See Data type, ARRAY
Multiplicative operator. See Operator, Multiplicative

N

Nested function. See Function, Nested
Nested procedure. See Procedure, Nested
Nested routine. See Procedure, Nested and Function, Nested
NEW procedure. See Procedure, NEW
New Project
  dialog box 24
NIL 481
Node 484
Null set 370, 377, 383

O

Object
  Ancestor 559
  Descendant 559
Object type 559
Object-oriented programming 559–560
ODD. See Function, ODD
OPEN procedure. See Procedure, OPEN
Operation
  for RECORDs 359
Operations
  on text files 425–426
Operator 79–83
  & 131
  | 134
  Additive 105, 378, 390
  Address 475
  AND 131
  Arithmetic 98
  Assignment 80, 95, 124
  Bitwise addition 538
  Bitwise AND 532
    Mask 535
  Bitwise exclusion 542
  Bitwise multiplication 532
  Bitwise negation 528
  Bitwise NOT 528
  Bitwise OR 538
  BOOLEAN 131–140
  Comparison 81–83, 95, 124, 130
  Concatenation 125
  Equality 81
  Exponentiation 102
  Greater than 82
  Greater than or equal to 82
  IN 372
  Inequality 81
  Left shift 543
  Less than 82
  Less than or equal to 82
  Multiplicative 105, 383
  NOT 138
  OR 134
  Pointer 475
  Right shift 547
  Set addition 374–378
  Set difference 383–390
  Set equality 390
  Set inequality 390
  Set intersection 378–383
  Set union 374–378
  Shift 543
  SHL 543
  SHR 547
  Subset 391
  Superset 391
  XOR 136, 542
    Toggling 542

- Operator precedence 83
  - for bitwise operators 542
  - for BOOLEANs 140
  - for REALs 105
  - Set difference 390
  - Set intersection 383
  - Set union 378
  - Shift 551
- Operators
  - for bit handling 528–551
  - for BOOLEANs 131–140
  - for CHARs 124
  - for SETs 371–393
- ORD. See Function, ORD
- Output
  - for BOOLEANs 131
- OVERRIDE statement. See Statement, OVERRIDE

P

- Parameter 34, 154–164
  - ARRAY 321–322
  - File 408
  - Formal 156
  - Self 584
  - Two-way 158
- Pascal 15
  - Predefined identifiers 76
  - Reserved keywords 76
  - Runtime library 517
  - Unit 517, 521–528
  - See also CodeWarrior Pascal
- Pascal program 70–72
  - Building an executable program 39
  - Changing 40
  - Comments 72
  - Components 32
  - Creating a project 36
  - Creating a source file 35
  - Execution 33
  - Structure 31
  - Style 33
- Pointer operator. See Operator, Pointer
- Pointer type. See Data type, Pointer
- Polymorphism 586
- POS. See Function, POS
- Precision
  - for type REAL 94
- PRED. See Function, PRED
- Predefined identifiers 76
- Procedure
  - Accessibility 286–300
  - APPEND 422, 425
  - Argument 156
  - ASSIGN 397, 398, 425
  - BlockRead 459
  - BlockWrite 458
  - Body 150
  - Call 146
  - Calling 275–277
  - CLOSE 400, 425
  - CYCLE 269
  - DEC 198
  - Declarations in 149
  - Definition 146
  - DELETE 200
  - DISPOSE 479
  - EXIT 273
  - General features 145
  - HALT 273
  - Heading 149
  - INC 201
  - INSERT 203
  - LEAVE 271
  - Nested 277–279
  - NEW 479
  - OPEN 397
  - Pascal 148
  - READ 88–90, 184, 425
    - for files 412
    - versus READLN 413–417
  - READLN 88–90, 184, 425
    - for files 412
    - versus READ 413–417
  - RESET 397, 398, 425
  - REWRITE 397, 398, 401, 425
  - SEEK 433, 437
  - SRAND 211
  - WRITE 85–88, 194, 425
    - for files 399
  - WRITELN 85–88, 194, 425
    - for files 399
- Procedure. See also Scope

Procedure call
  Flow 303
Procedures
  Input 88–90
  Output 85–88
    for files 399
Programming style
  Object-oriented 553, 559–560
  Traditional 553
Project 18
  Creating 24, 36
  Starting 36
Project Settings
  Dialog box 20

R

RAND. See Function, RAND
Random access structure 411
READ. See Procedure, READ
READLN. See Procedure, READLN
Real number 90–105
Real number type, see Data type, Real number
REAL type. See Data type, REAL
RECORD type. See Data type, RECORD
RECORD type. See Type, RECORD
Record variant 359–364
Recursion 300–311
  Features 304
  Flow 303
REPEAT Statement. See Statement, REPEAT
RESET procedure. See Procedure, RESET
REWRITE procedure. See Procedure, REWRITE
Right shift operator. See Operator, Right shift
ROUND. See Function, ROUND
Routine
  Global 296
  Local 296
Runtime library 517

S

Scientific notation 92
Scope 296–297
  Applicability 296
SEEK procedure. See Procedure, SEEK
Self 584
Self-referential structure 483–515
Sequential access structure 411
Set addition operator. See Operator, Set addition
Set constructor 368
Set difference operator. See Operator, Set difference
Set intersection operator. See Operator, Set intersection
SET type. See Type, SET
Set union operator. See Operator, Set union
Shift operator. See Operator, Shift
SHL operator. See Operator, SHL
SHR operator. See Operator, SHR
Simple Statement. See Statement, Simple
SIN. See Function, SIN
SINGLE type. See Data type, SINGLE
SIZEOF. See Function , SIZEOF
Source file
  Editing 41
  Including during compilation 285
SQR. See Function, SQR
SQRT. See Function, SQRT
SRAND. See Procedure, SRAND
Stack segment 461
Statement 34, 70
  CASE 246
  Compound 35
  FOR 215
  IF 226
  IF-ELSE 229
  OVERRIDE 574
  REPEAT 253
    Structure 262
    versus WHILE 267
  Simple 34
  USES 522
  WHILE 257
    Structure 262
    versus REPEAT 267
  WITH 358
Static memory allocation 461–462
Stationery 36
STRING array. See Data type, ARRAY
STRING type. See Data type, STRING
Structure

- Random access 411
- Sequential access 411

Structured type. See Type, Structured

Subset 390

SUCC.. See Function, SUCC

Superset 390

T

Tag Field 362

Target variable
- Accessing 468

Termination condition 256

Text file
- Operations on 425–426

TEXT type. See Data type, TEXT

Toggling 542

Tree 484

TRUNC. See Function, TRUNC

Type
- Aggregate 313
- ARRAY 313, 315–341
  - Defining 316–318
  - Parameter 321–322
- RECORD 313
- SET 313
- Structured 313

TYPE definition 142

Typed file 426–450
- Defining 430

Types.P 521

U

Union operator 374–378

Unit 517, 517–528
- Creating 524
- Implementation 520, 526
- Interface 517

UNSIGNEDLONG type. See Data type, UNSIGNEDLONG

UNSIGNEDWORD type. See Data type, UNSIGNEDWORD

Untyped file 454–459

UPCASE. See Function, UPCASE

USES statement. See Statement, USES

V

Variable 74–79
- Accessibility 286–300
- Declaration 77
- Enumerated 143
- Global versus local 150
- Identifier 74
- Initialization 79, 85
- Loop 218
- Name 74
- Name restrictions 76
- Naming 74
- Target
  - Accessing 468
  - versus pointer 470

Variable. See also Scope

Variables.P 521

Visibility 297–300

W

WHILE Statement. See Statement, WHILE

Whole number 106–121

Windows.P 521

Wirth, Niklaus 15

WITH 348–354
- Independent 352
- Nested 353

WRITE. See Procedure, WRITE

WRITELN. See Procedure, WRITELN

X

XOR operator. See Operator, XOR

CodeWarrior

Learn Pascal Under Windows 95/NT

Credits

writing lead: You
other writers: Your co-workers
engineering: Dilbert
frontline warriors: The folks that helped a lot

Guide to CodeWarrior Documentation

If you need information about...	See this
Installing updates to CodeWarrior	*QuickStart Guide*
Getting started using CodeWarrior	*QuickStart Guide*; *Tutorials* (Apple Guide)
Using CodeWarrior IDE (Integrated Development Environment)	*IDE User's Guide*
Debugging	*Debugger Manual*
Important last-minute information on new features and changes	*Release Notes* folder
Creating Macintosh and Power Macintosh software	*Targeting Mac OS*; *Mac OS* folder
Creating Microsoft Win32/x86 software	*Targeting Win32*; *Win32/x86* folder
Creating Java software	Targeting Java Sun Java Documentation folder
Creating Magic Cap software	*Targeting Magic Cap*; *Magic Cap* folder
Using ToolServer with the CodeWarrior editor	*IDE User's Guide*
Controlling CodeWarrior through AppleScript	*IDE User's Guide*
Using CodeWarrior to program in MPW	*Command Line Tools Manual*
C, C++, or 68K assembly-language programming	*C, C++, and Assembly Reference*; *MSL C Reference*; *MSL C++ Reference*
Pascal or Object Pascal programming	*Pascal Language Manual*; *Pascal Library Reference*
Fixing compiler and linker errors	*Errors Reference*
Fixing memory bugs	*ZoneRanger Manual*
Speeding up your programs	*Profiler Manual*
PowerPlant	*The PowerPlant Book;* PowerPlant Advanced Topics; PowerPlant reference documents
Creating a PowerPlant visual interface	*Constructor Manual*
Creating a Java visual interface	Constructor for Java Manual
Learning how to program for the Mac OS	*Discover Programming for Macintosh*
Learning how to program in Java	Discover Programming for Java
Contacting Metrowerks about registration, sales, and licensing	*Quick Start Guide*
Contacting Metrowerks about problems and suggestions using CodeWarrior software	*email Report Forms* in the *Release Notes* folder
Sample programs and examples	*CodeWarrior Examples* folder; *The PowerPlant Book*; *PowerPlant Advanced Topics;* *Tutorials* (Apple Guide)
Problems other CodeWarrior users have solved	*Internet newsgroup [docs]* folder

Revision code